推酷

编程狂人

Programming Madman

NO.26

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:

http://www.tuicool.com/mags/53828d44d91b14075d0805b9

欢迎下载推酷客户端体验更多阅读乐趣

目录

JavaScript装逼指南

作者: civerzhu

如何写JavaScript才能逼格更高呢？怎样才能组织JavaScript才能让别人一眼看出你不简单呢？是否很期待别人在看完你的代码之后感叹一句“原来还可以这样写”呢？下面列出一些在JavaScript时的装逼技巧。

1. 匿名函数的N种写法

你知道“茴”的四种写法吗？ε=(·д· `*)ハァ… 扯淡，但你或许不知道匿名函数的好几种写法。一般情况下写匿名函数是这样的：

(function(){})();

但下面几种写法也是可以的：

- ***!function(){}();***
- ***+function(){}();***
- ***-function(){}();***
- ***~function(){}();***
- ***~(function(){})();***
- ***void function(){}();***
- ***(function(){}());***

当然，这样的写法，没有什么区别，纯粹看装逼程度。

2. 另外一种undefined

从来不需要声明一个变量的值是undefined，因为JavaScript会自动把一个未赋值的变量置为undefined。所有如果你在代码里这么写，会被鄙视的：

var data = undefined;

但是如果你就是强迫症发作，一定要再声明一个暂时没有值的变量的时候赋上一个undefined。那你可以考虑这么做：

var data = void 0; // undefined

void在JavaScript中是一个操作符，对传入的操作不执行并且返回undefined。void后面可以跟()来用，例如void(0)，看起来是不是很熟悉？没错，在HTML里阻止带href的默认点击操作时，都喜欢把href写成javascript:void(0)，实际上也是依靠void操作不执行的意思。

当然，除了出于装逼的原因外，实际用途上不太赞成使用void，因为void的出现是为了兼容早起ECMAScript标准中没有undefined属性。void 0的写法让代码晦涩难懂。

3. 抛弃你的if和else

当JS代码里有大量的条件逻辑判断时，那代码看起来多可怕：

```
if () {
    // ...
} else if () {
    // ...
} else if () {
    // ...
} else {
    // ...
}
```

不用我说你都猜到用什么语法来简化if-else了。没错，用||和&&，很简单的原理就不用说啦。值得一提的是，有时候用!!操作符也能简化if-else模式。例如这样：

```
// 普通的if-else模式
var isValid = false;
if (value && value !== 'error') {
    isValid = true;
}
// 使用!!符号
var isValid = !!(value && value !== 'error');
```

“!”是取反操作，两个“!”自然是负负得正了。

4. 不加分号

关于JavaScript要不要加分号的争论已经吵了好几年。Google的JavaScript语法指南告诉我们要加分号，很多JavaScript语法书籍也告诉我们加上分号更安全。然而，分号加不加，全靠个人对代码的写法，你确信写得足够安全的话，不加分号显得更加高大上。

5. 赶上ES6的早班车

ES6即将在年底正式发布，赶时髦的开发者们，赶快在自己的代码里用起来。用上module声明，写写class，捣鼓一下Map，这些都会让你的代码逼格更高。神马？你都不会用？那也好歹在代码头部加上一个ES5的"use strict";呀。

6. 添加AMD模块支持

给你写的代码声明一下AMD模块规范，这样别人就可以直接通过AMD的规范来加载你的模块了，如果别人没有通过规范来加载你的模块，你也可以优雅地返回一个常规的全局对象。来看看jQueryUI的写法：

```
(function( factory ) {
```

```
    if ( typeof define === "function" && define.amd ) {
        // AMD. Register as an anonymous module.
        define( [ "jquery" ], factory );
    } else {
        // Browser globals
        factory( jQuery );
    }
}(function( $ ) {
    // 这里放模块代码
    return $.widget;
}));
```

就用它来包裹你的实际代码吧，保证别人一看代码就知道你是个专业的开发者。

7. Function构造函数

很多JavaScript教程都告诉我们，不要直接用内置对象的构造函数来创建基本变量，例如var arr = new Array(2); 的写法就应该用var arr = [1, 2];的写法来取代。但是，Function构造函数（注意是大写的Function）有点特别。Function构造函数接受的参数中，第一个是要传入的参数名，第二个是函数内的代码（用字符串来表示）。

```
var f = new Function('a', 'alert(a)');

f('test'); // 将会弹出窗口显示test
```

或许大家疑惑了，你这样绕着写，跟function f(a) {alert(a);}比有什么好处呢?

事实上在某种情况下是有好处的，比如不能用eval的时候，你需要传入字符串内容来创建一个函数的时候。在一些JavaScript模板语言的解析，和字符串转换json对象的时候比较实用。

8. 用原生Dom接口不用jQuery

一个傲娇的前端工程师是不需要jQuery的，前提是你经得起折腾。实际上，几乎所有的jQuery方法都可以用同样的Dom原生接口来实现，因为这货本来就是用原生接口实现的嘛，哈哈。怎样做到不用jQuery（也叫jQuery-free）呢？阮老师的博文《如何做到 jQuery-free？》 给我们很好的讲解了做法。依赖于querySelector和querySelectorAll这两个现代浏览器的接口，可以实现跟jQuery同样方便和同样效率的Dom查找，而且其他的类似Ajax和CSS的接口同样也可以把原生方法做一些兼容方面的包装即可做到jQuery-free。

总结

上述所有的JavaScript装逼写法，一些是为了程序易懂或者效率提高的语法糖，这样的做法是比较可取的，比如前面所说的省略if-else的做法；而有些写法则容易造成代码晦涩难懂或者效率偏低，例如上面说的void 0的写法，实际上不可取。JavaScript语法上灵活，让大家对同一个功能有很多种不同的写法，写法上的优化是对程序结构和代码维护有很大帮助的。所以，装逼得装得好看。

原文链接:http://blog.segmentfault.com/civerzhu/1190000000514581

探索Javascript异步编程

作者: 刚

笔者在之前的一片博客中简单的讨论了Python和Javascript的异同，其实作为一种编程语言Javascript的异步编程是一个非常值得讨论的有趣话题。

JavaScript 异步编程简介

回调函数和异步执行

所谓的异步指的是函数的调用并不直接返回执行的结果，而往往是通过回调函数异步的执行。

我们先看看回调函数是什么:

```
var fn = function(callback) {
    // do something here
    ...
    callback.apply(this, para);
};

var mycallback = function(parameter) {
    // do someting in customer callback
};
```

```
// call the fn with callback as parameter
fn(mycallback);
```

回调函数，其实就是调用用户提供的函数，该函数往往是以参数的形式提供的。回调函数并不一定是异步执行的。比如上述的例子中，回调函数是被同步执行的。大部分语言都支持回调，C++可用通过函数指针或者回调对象，Java一般也是使用回调对象。

在Javascript中有很多通过回调函数来执行的异步调用，例如setTimeout（）或者setInterval（）。

```
setTimeout(function(){
    console.log("this will be exectued after 1 second!");
},1000);
```

在以上的例子中，setTimeout直接返回，匿名函数会在1000毫秒（不一定能保证是1000毫秒）后异步触发并执行，完成打印控制台的操作。也就是说在异步操作的情境下，函数直接返回，把控制权交给回调函数，回调函数会在以后的某一个时间片被调度执行。那么为什么需要异步呢？为什么不能直接在当前函数中完成操作呢？这就需要了解Javascript的线程模型了。

Javascript线程模型和事件驱动

Javascript最初是被设计成在浏览器中辅助提供HTML的交互功能。在浏览器中都包含一个Javascript引擎，Javscript程序就运行在这个引擎之中，并且只有一个线程。单线程能都带来很多优点，程序员们可以很开心的不用去考虑诸如资源同步，死锁等多线程阻塞式编程所需要面对的恼人的问题。但是很多人会问，既然Javascript是单线程的，那它又如何能够异步的执行呢？

这就需要了解到Javascript在浏览器中的事件驱动（event driven）机制。事件驱动一般通过事件循环（event loop）和事件队列（event

queue）来实现的。假定浏览器中有一个专门用于事件调度的实例（该实例可以是一个线程，我们可以称之为事件分发线程event dispatch thread），该实例的工作就是一个不结束的循环，从事件队列中取出事件，处理所有很事件关联的回调函数（event handler）。注意回调函数是在Javascript的主线程中运行的，而非事件分发线程中，以保证事件处理不会发生阻塞。

Event Loop Code:

```
while(true) {
 var event = eventQueue.pop();
 if(event && event.handler) {
    event.handler.execute(); // execute the callback in Javascript thread
 } else {
    sleep(); //sleep some time to release the CPU do other stuff
 }
}
```

通过事件驱动机制，我们可以想象Javascript的编程模型就是响应一系列的事件，执行对应的回调函数。很多UI框架都采用这样的模型（例如Java Swing）。

那为什要异步呢，同步不是很好么?

异步的主要目的是处理非阻塞，在和HTML交互的过程中，会需要一些IO操作（典型的就是Ajax请求，脚本文件加载），如果这些操作是同步的，就会阻塞其它操作，用户的体验就是页面失去了响应。

综上所述Javascript通过事件驱动机制，在单线程模型下，以异步回调函数的形式来实现非阻塞的IO操作。

Javascript异步编程带来的挑战

Javascript的单线程模型有很多好处，但同时也带来了很多挑战。

代码可读性

想象一下，如果某个操作需要经过多个非阻塞的IO操作，每一个结果都是通过回调，程序有可能会看上去像这个样子。

```
operation1(function(err, result) {
    operation2(function(err, result) {
        operation3(function(err, result) {
            operation4(function(err, result) {
                operation5(function(err, result) {
                    // do something useful
                })
            })
        })
    })
})
```

我们称之为意大利面条式（spaghetti）的代码。这样的代码很难维护。这样的情况更多的会发生在server side的情况下。

流程控制

异步带来的另一个问题是流程控制，举个例子，我要访问三个网站的内容，当三个网站的内容都得到后，合并处理，然后发给后台。代码可以这样写:

```
var urls = ['url1','url2','url3'];
var result = [];

for (var i = 0, len = urls.length(); i < len; i++ ) {
    $.ajax({
        url: urls[i],
        context: document.body,
        success: function(){
          //do something on success
          result.push("one of the request done successfully");
          if (result.length === urls.length()) {
              //do something when all the request is completed successfully
          }
        });
    }
```

上述代码通过检查result的长度的方式来决定是否所有的请求都处理完成，这是一个很丑陋方法，也很不可靠。

异常和错误处理

通过上一个例子，我们还可以看出，为了使程序更健壮，我们还需要加入异常处理。 在异步的方式下，异常处理分布在不同的回调函数中，我们无法在调用的时候通过try...catch的方式来处理异常， 所以很难做到有效，清楚。

更好的Javascript异步编程方式

“这是最好的时代，也是最糟糕的时代”

为了解决Javascript异步编程带来的问题，很多的开发者做出了不同程度的努力，提供了很多不同的解决方案。然而面对如此众多的方案应该如何选择呢？我们这就来看看都有哪些可供选择的方案吧。

Promise

Promise 对象曾经以多种形式存在于很多语言中。这个词最先由C++工程师用在Xanadu 项目中，Xanadu 项目是Web 应用项目的先驱。随后Promise 被用在E编程语言中，这又激发了Python 开发人员的灵感，将它实现成了Twisted 框架的Deferred 对象。

2007 年，Promise 赶上了JavaScript 大潮，那时Dojo 框架刚从Twisted 框架汲取灵感，新增了一个叫做dojo.Deferred 的对象。也就在那个时候，相对成熟的Dojo 框架与初出茅庐的jQuery 框架激烈地争夺着人气和名望。2009 年，Kris Zyp 有感于dojo.Deferred 的影响力提出了CommonJS 之Promises/A 规范。同年，Node.js 首次亮相。

在编程的概念中，future，promise，和delay表示同一个概念。Promise翻译成中文是“承诺”，也就是说给你一个东西，我保证未来能够做到，但现在什么都没有。它用来表示异步操作返回的一个对象，该对象是用来获取未来的执行结果的一个代理，初始值不确定。许多语言都有对Promise的支持。

Promise的核心是它的then方法，我们可以使用这个方法从异步操作中得到返回值，或者是异常。then有两个可选参数（有的实现是三个），分别处理成功和失败的情景。

```
var promise = doSomethingAync()

promise.then(onFulfilled, onRejected)
```

异步调用doSomethingAync返回一个Promise对象promise，调用promise的then方法来处理成功和失败。这看上去似乎并没有很大的改进。仍然需要回调。但是和以前的区别在于，首先异步操作有了返回值，虽然该

值只是一个对未来的承诺；其次通过使用then，程序员可以有效的控制流程异常处理，决定如何使用这个来自未来的值。

对于嵌套的异步操作，有了Promise的支持，可以写成这样的链式操作：

```
operation1().then(function (result1) {
    return operation2(result1)
}).then(function (result2) {
    return operation3(result2);
}).then(function (result3) {
    return operation4(result3);
}).then(function (result4) {
    return operation5(result4)
}).then(function (result5) {
    //And so on
});
```

Promise提供更便捷的流程控制，例如Promise.all()可以解决需要并发的执行若干个异步操作，等所有操作完成后进行处理。

```
var p1 = async1();
var p2 = async2();
var p3 = async3();
Promise.all([p1,p2,p3]).then(function(){
    // do something when all three asychronized operation finished
});
```

对于异常处理，

```
doA()
  .then(doB)
  .then(null,function(error){
      // error handling here
  })
```

如果doA失败，它的Promise会被拒绝，处理链上的下一个onRejected会被调用，在这个例子中就是匿名函数function（error）{}。比起原始的回调方式，不需要在每一步都对异常进行处理。这生了不少事。

以上只是对于Promise概念的简单陈述，Promise拥有许多不同规范建议（A,A+,B,KISS,C,D等），名字（Future，Promise，Defer），和开源实现。大家可以参考一下的这些链接。

- jQuery's Deferred Object

http://api.jquery.com/category/deferred-object/

- YUI Promise Class

http://yuilibrary.com/yui/docs/promise/

- Dojo Promises

http://dojotoolkit.org/documentation/tutorials/1.6/promises/

- Q

http://documentup.com/kriskowal/q

• RSVP.js

https://github.com/tildeio/rsvp.js

• When.js

https://github.com/cujojs/when

• MochiKit.Async

http://mochi.github.io/mochikit/doc/html/MochiKit/Async.html

• FutureJS

https://github.com/FuturesJS

• node-promise

https://github.com/kriszyp/node-promise

• WinJS

http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx

如果你有选择困难综合症，面对这么多的开源库不知道如何决断，先不要急，这还只是一部分，还有一些库没有或者不完全采用Promise的概念

Non-Promise

下面列出了其它的一些开源的库，也可以帮助解决Javascript中异步编程所遇到的诸多问题，它们的解决方案各不相同，我这里就不一一介绍了。大家有兴趣可以去看看或者试用一下。

• Node-fibers

https://github.com/laverdet/node-fibers

• Streamlinejs

https://github.com/Sage/streamlinejs

• Step

https://github.com/creationix/step

• Flow-js

https://github.com/willconant/flow-js

• Async

https://github.com/caolan/async

• Async.js

https://github.com/fjakobs/async.js

• slide-flow-control

https://github.com/isaacs/slide-flow-control

Non-3rd Party

其实，为了解决Javascript异步编程带来的问题，不一定非要使用Promise或者其它的开源库，这些库提供了很好的模式，但是你也可以通过有针对性的设计来解决。

比如，对于层层回调的模式，可以利用消息机制来改写，假定你的系统中已经实现了消息机制，你的code可以写成这样：

```
eventbus.on("init", function(){
    operationA(function(err,result){
        eventbus.dispatch("ACompleted");
    });
});

eventbus.on("ACompleted", function(){
    operationB(function(err,result){
        eventbus.dispatch("BCompleted");
```

```
    });
});

eventbus.on("BCompleted", function(){
    operationC(function(err,result){
        eventbus.dispatch("CCompleted");
    });
});

eventbus.on("CCompleted", function(){
    // do something when all operation completed
});
```

这样我们就把嵌套的异步调用，改写成了顺序执行的事件处理。

更多的方式，请大家参考这篇文章，它提出了解决异步的五种模式：回调、观察者模式（事件）、消息、Promise和有限状态机（FSM）。

下一代Javscript对异步编程的增强

ECMAScript6

下一代的Javascript标准Harmony，也就是ECMAScript6正在酝酿中，它提出了许多新的语言特性，比如箭头函数、类（Class）、生成器（Generator）、Promise等等。其中Generator和Promise都可以被用于对异步调用的增强。

Nodejs的开发版V0.11已经可以支持ES6的一些新的特性，使用node --harmony命令来运行对ES6的支持。

co、Thunk、Koa

koa是由Express原班人马（主要是TJ）打造，希望提供一个更精简健壮的nodejs框架。koa依赖ES6中的Generator等新特性，所以必须运行在相应的Nodejs版本上。

利用Generator、co、Thunk，可以在Koa中有效的解决Javascript异步调用的各种问题。

co是一个异步流程简化的工具，它利用Generator把一层层嵌套的调用变成同步的写法。

```
var co = require('co');
var fs = require('fs');

var stat = function(path) {
  return function(cb){
    fs.stat(path,cb);
  }
};

var readFile = function(filename) {
  return function(cb){
    fs.readFile(filename,cb);
  }
};

co(function *() {
  var stat = yield stat('./README.md');
```

```
  var content = yield readFile('./README.md');
})();
```

通过co可以把异步的fs.readFile当成同步一样调用，只需要把异步函数fs.readFile用闭包的方式封装。

利用Thunk可以进一步简化为如下的code, 这里Thunk的作用就是用闭包封装异步函数，返回一个生成函数的函数，供生成器来调用。

```
var thunkify = require('thunkify');
var co = require('co');
var fs = require('fs');

var stat = thunkify(fs.stat);
var readFile = thunkify(fs.readFile);

co(function *() {
  var stat = yield stat('./README.md');
  var content = yield readFile('./README.md');
})();
```

利用co可以串行或者并行的执行异步调用。

串行

```
co(function *() {
  var a = yield request(a);
  var b = yield request(b);
```

```
})();
```

并行

```
?
1
2
3
co(function *() {
 var res = yield [request(a), request(b)];
})();
```

更多详细的内容，大家可以参考这两篇文章1，2。

总结

异步编程带来的问题在客户端Javascript中并不明显，但随着服务器端Javascript越来越广的被使用，大量的异步IO操作使得该问题变得明显。许多不同的方法都可以解决这个问题，本文讨论了一些方法，但并不深入。大家需要根据自己的情况选择一个适于自己的方法。

同时，随着ES6的定义，Javascript的语法变得越来越丰富，更多的功能带来了很多便利，然而原本简洁，单一目的的Javascript变得复杂，也要承担更多的任务。Javascript何去何从，让我们拭目以待。

原文链接:http://gangtao.is-programmer.com/posts/46795.html

C++插件中使用静态指针变量引起的内存泄露问题

作者: 五竹

在C++的动态库中，有是为了实现Singleton等功能，经常会使用静态(static)指针变量，并在第一次使用是申请动态分配对象(new)； 但其内存的释放往往依赖程序退出时，操作系统来完成内存回收。对于一般的应用，这是没有问题的，但对于C++ 的插件来说，因为其可能在服务程序中被动态的热加载/卸载(dlopen/dlclose)，此时，往往会带来内存泄露问题。

下面来看个示例，来说明这种情况：这个例子中，在插件中，声明一个静态成员指针变量 _buff, 并在第1次使用是申请内存10M. 把代码编译成2个功能相同的动态库（插件），然后各 dlopen/dlclose 50次，看看程序的内存使用情况。

- ***DsoBase.h***

```
#ifndef __DSO_BASE_H_
#define __DSO_BASE_H_
namespace name_1
{
class DsoBase
{
public:
    DsoBase(){};
```

```
    virtual ~DsoBase(){};
    virtual void toString()=0;
};
}
#endif
```

- **Dso.h**

```
#ifndef __DSO_DSO_H_
#define __DSO_DSO_H_
#include <stdint.h>
#include <string>
#include "DsoBase.h"

namespace name_1
{
class Dso:public DsoBase
{
public:
    Dso() ;
    virtual ~Dso() ;
    virtual void toString();
```

```
private:
    static pthread_mutex_t _mutex;
    static char * _buff;    // 静态成员变量
    const  uint32_t MAX_LEN;
};
}
#endif
```

- **Dso.cpp**

```
#include <stdio.h>
#include "Dso.h"

namespace name_1
{
pthread_mutex_t Dso::_mutex = PTHREAD_MUTEX_INITIALIZER;
char * Dso::_buff=NULL;   // 静态成员变量
Dso::Dso():MAX_LEN(10*1024*1024)
{
    printf("Dso constructor\n");
}
Dso::~Dso()
{
    printf("Dso destructor\n");
```

```
}

void Dso::toString()
{
    pthread_mutex_lock(&_mutex);
    if(_buff == NULL)
    {//第1次使用是申请分配10M内存
        _buff=(char*)malloc(MAX_LEN);
        memset(_buff,0x0,MAX_LEN);
        printf("_buff=%p\n",_buff);
    }
    pthread_mutex_unlock(&_mutex);
}

}

extern "C" name_1::DsoBase * CreateFun()
{
    return  new name_1::Dso();
}
extern "C" void DestroyFun(name_1::DsoBase * obj)
{
    delete obj;
```

```
}
```

- **test_main.cpp**

```
#include <stdio.h>
#include <dlfcn.h>
#include "pthread.h"
#include "Dso.h"

typedef name_1::DsoBase * (*CreateFunT)();
typedef void (*DestroyFunT)(name_1::DsoBase * p);

int main(int argc , char ** argv)
{

    // 模拟 ha3 插件更新过程
    for(int i=0; i <50; i ++ )
    {
        //1. dl_open libdso_a.so
      void * dl_a=dlopen("./libdso_a.so",RTLD_NOW);
      if(!dl_a ) {
        printf("dlopen return %s", dlerror());
        return -1;
      }
```

```
        CreateFunT createFun_a= (CreateFunT)dlsym(dl_a , "Create-
Fun");
        DestroyFunT destoryFunc_a = (DestroyFunT)dlsym(dl_a , "De-
stroyFun");
        name_1::DsoBase* obj_a=createFun_a();
        obj_a->toString();

        //2. dl_open libdso_b.so
        void * dl_b=dlopen("./libdso_b.so",RTLD_NOW);
        if(!dl_b ) {
            printf("dlopen return %s", dlerror());
            return -1;
        }
        CreateFunT createFun_b= (CreateFunT)dlsym(dl_b , "Create-
Fun");
        DestroyFunT destoryFunc_b = (DestroyFunT)dlsym(dl_b , "De-
stroyFun");
        name_1::DsoBase* obj_b=createFun_b();
        obj_b->toString();
        sleep(5);
       //3. dl_close libdso_a.so
        destoryFunc_a(obj_a);
        dlclose(dl_a);
        sleep(5);
       //4. 使用 obj_b
```

```
        obj_b->toString();

       //5. dl_close libdso_a.so
        destoryFunc_b(obj_b);
        dlclose(dl_b);
        sleep(20);
    }


    return 0;
}
```

- **Scons 的 SContruct**

```
# -*- mode: python -*-

import sys, os, os.path, platform, re, time

env = Environment()

env.AppendUnique(CCFLAGS = '-g')
env.AppendUnique(CCFLAGS = '-m64')
env.AppendUnique(CCFLAGS = '-DTARGET_64')
env.AppendUnique(LINKFLAGS = '-m64')
```

dso_sources = ['Dso.cpp']

env.SharedLibrary('dso_a', dso_sources)

env.SharedLibrary('dso_b', dso_sources)

env.Program('test_main',['test_main.cpp'], LIBS=['dl'], LIBPATH='.',LINKFLAGS = '-export-dynamic')

- 编译

[static_var_and_dynamic_lib_t2]$scons -c && scons

scons: Reading SConscript files …

scons: done reading SConscript files.

scons: Cleaning targets …

Removed Dso.os

Removed libdso_a.so

Removed libdso_b.so

Removed test_main.o

Removed test_main

scons: done cleaning targets.

scons: Reading SConscript files …

scons: done reading SConscript files.

scons: Building targets …

g++ -o Dso.os -c -g -m64 -DTARGET_64 -fPIC Dso.cpp

g++ -o libdso_a.so -m64 -shared Dso.os

g++ -o libdso_b.so -m64 -shared Dso.os

g++ -o test_main.o -c -g -m64 -DTARGET_64 test_main.cpp

g++ -o test_main -export-dynamic test_main.o -L. -ldl

scons: done building targets

• 执行测试程序

[static_var_and_dynamic_lib_t2]$./test_main

Dso constructor

_buff=0x2b65d7f31010

Dso constructor

_buff=0x2b65d8b34010

Dso destructor

Dso destructor

Dso constructor

_buff=0x2b65d9535010

Dso constructor

_buff=0x2b65da138010

Dso destructor

Dso destructor

Dso constructor

_buff=0x2b65dab39010

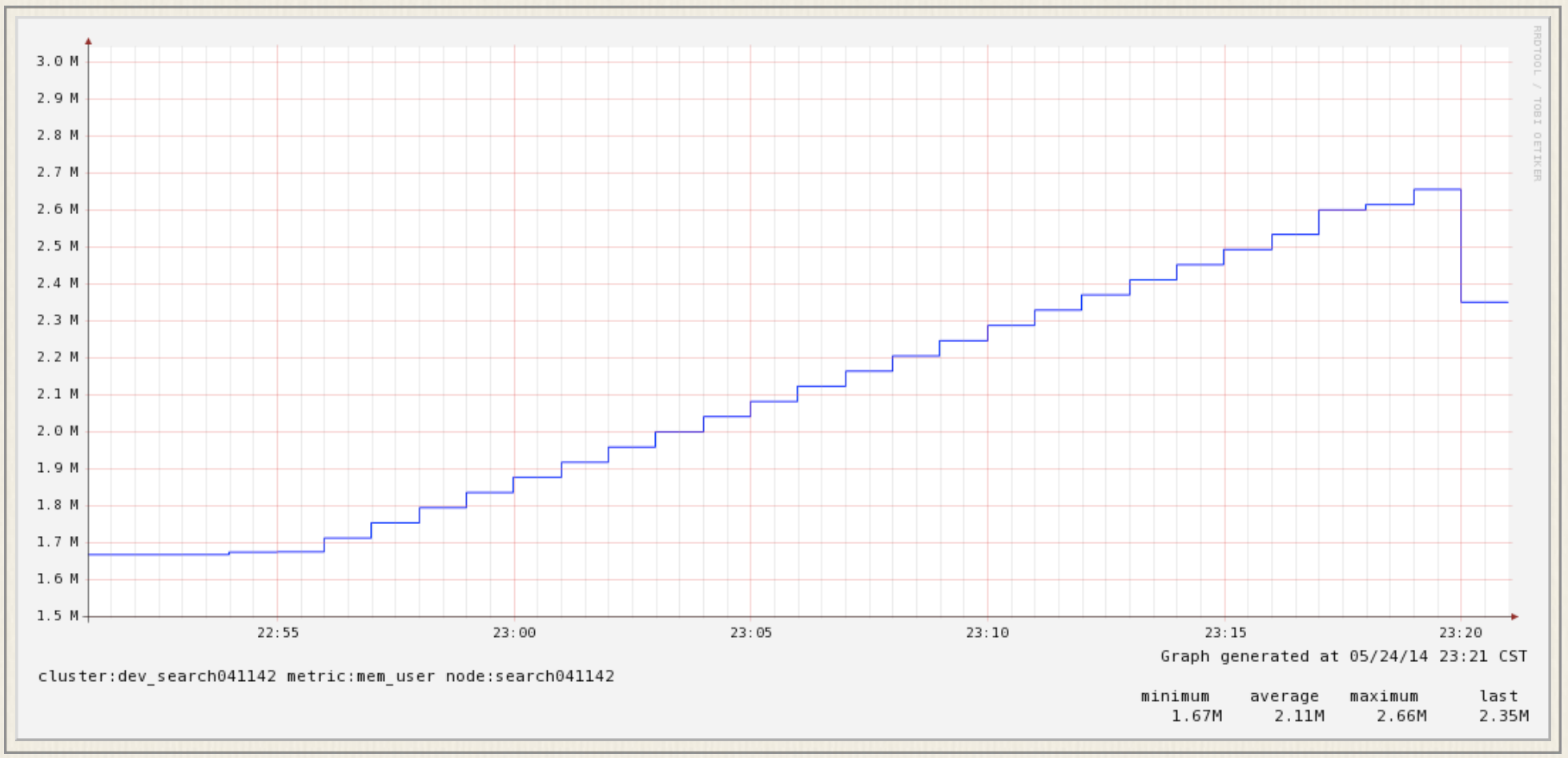
Dso constructor

…

- 内存使用情况—*mem_user*

从程序开始执行到结束， 系统 mem_user 从 1.7G (图中1M 表示内存1G)涨到近 2.7G,左右，增涨1G， 正好是2个插件（libdso_a.so 和 libdso_b.so），50次dlopen/dlclose，每次申请 10M 的内存累积量。

可见 dlclose 时，并不会释放动态库(即插件)内动态申请的内存，所以引起内存泄露。



原文链接:http://www.searchtb.com/2014/05/c插件中使用静态指针变量引起的内存泄露问题.html

一个用户迁移数据库前后的性能差异case

作者: 丁林.tb

问题

一个用户问题，数据从ECS迁移到RDS，相同的语句，查询性能下降了几十倍。而实际上RDS这个实例在内存上的配置与原来ECS上的实例相当。

本文简单说明这个case的原因及建议。

用户反馈性能变慢的语句为 （修改了真实表名和列名）

```
select count(1)  from HR hr join H h on h.hid = hr.hid
join A e on  e.aid = h.eid
join A t on t.aid = e.pid
join A c on c.aid = t.pid
join A p on p.aid = c.pid
left join U u on u.uid = hr.uld
left join E emp on emp.eid = hr.oid
where  ( hr.s in (1,2,3,4)  and hr.cn = 0 );
```

背景

MySQL执行语句过程中涉及到两大流程：优化器和执行器。其中优化器最主要的任务，是选择索引和在多表连接时选择连接顺序。在这个case中，join顺序的选择影响了执行性能。

确定join执行顺序就需要估算所有join操作的代价。默认配置下MySQL会估算所有可能的组合。

MySQL Tips: MySQL里限制一个查询的join表数目上限为61.

对于一个有61个表参与的join操作，理论上需要61!(阶乘)次的评估。当然这是最坏情况下，实际上减枝算法会让这个数字看起来稍微好一点，但是仍然很恐怖。

在多表join的场景下，为了避免优化器占用太多时间，MySQL提供了一个参数 optimizer_search_depth 来控制递归深度。

这个参数对算法的控制可以简单描述为：对于所有的排列，只取前当前join顺序的前optimizer_search_depth个表估算代价。举例来说，20张表的，假设optimizer_search_depth为4，那么评估次数为20*19*18*17，虽然也很大（因此我们特别不建议这么多表的join），比20！好多了。

于是optimizer_search_depth的选择就成了问题。

MySQL Tips: MySQL中optimizer_search_depth默认值为62.也就是说默认为全排列计算。

这样能够保证得到最优的执行计划，只是在有些场景下，决定执行计划的时间会远大于执行时间本身。

量化分析

在ECS上，是用户自己维护的MySQL，没有设置optimizer_search_depth，因此为默认的62.
在RDS上，我们的配置是4。
分析到这里大家能猜到原因是RDS配置的4导致没有得到最优的执行计划。

下图是optimizer_search_depth=4时的explain结果（隐藏了业务相关的表名、字段名)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	e	index		parentId	4	NULL	22039	Using index
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where; Using index
1	SIMPLE		ALL		NULL	NULL		82720	Using where; Using join buffer
1	SIMPLE		eq_ref	PRIMARY	PRIMARY	4		1	Using index
1	SIMPLE		eq_ref	PRIMARY	PRIMARY	4		1	Using index
1	SIMPLE	h	eq_ref	PRIMARY	PRIMARY	4		1	Using where

下图是optimizer_search_depth=62是的场景，当然这个case的join表是8个，因此62和8在这里是等效的。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE		ref		checkNum	4	const	41360	Using where
1	SIMPLE		eq_ref		PRIMARY	4		1	Using index
1	SIMPLE		eq_ref		PRIMARY	4		1	Using index
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where
1	SIMPLE		eq_ref		PRIMARY	4		1	Using where
1	SIMPLE		eq_ref	PRIMARY	PRIMARY	4		1	Using where; Using index

从图1可以看到，由于optimizer_search_depth=4，优化器认为自己选择了最优的join顺序(22039*1*1*1)，优于(41360*1*1*1)，而实际上后者才是全局最优。

有趣的是，在这个case里面如果多看一层，就能得到最有解，因为第一个join顺序的第五个表评估rows为82720。

这意味着，在这个case里面，设置为5与设置为62能得到相同的执行计划，当然设置为5时的优化器执行代价更小。这其实也就是提供optimizer_search_depth的本意：减少优化器执行时间，而且概率上还存在局部最优就是全局最优解的情况。

关于实践

可配置的参数提供灵活性的同时，也提出一个头疼的问题：应该设置为多少才合适。
实际上当用户执行一个多表join的时候，对这个语句的整体RT的期望值就不会高。因此可以先定义一个预期，比如优化器决策join顺序的时间不能超过500ms。
用户规格与cpu相关，因此这个只能是建议值。

用户实践

实际上更重要的是对于用户来说：

1) 当出现实例迁移后，多表join执行结果差异较大的时候，要考虑调整这个值。该参数是允许线程单独设置，因此对于应用层来说，每个连接应该都能得到一个较优的值。

2) 反过来，当设置为默认的optimizer_search_depth=62时，我们我们如何评估我们这个设置是否过大?

MySQL Tips:MySQL profiling 可以用于查看各执行环节的消耗时间。

如下是笔者构造的一个60个表join查询的查询，使用profiling查看执行环节消耗的过程。

```
set profiling=1;

set optimizer_search_depth=4;
```

explain select .....

show profile for query 2;

结果如图

Opening tables	0.000172
System lock	0.000056
init	0.000221
optimizing	0.000095
statistics	0.005288
preparing	0.000138
executing	0.000190
end	0.000010
query end	0.000012
closing tables	0.000039
freeing items	0.000038
logging slow query	0.000002
cleaning up	0.000005

继续执行

set optimizer_search_depth=40;

explain select .....

show profile for query 4;

Opening tables	0.000141
System lock	0.000049
init	0.000217
optimizing	0.000098
statistics	0.915351
preparing	0.000138
executing	0.000226
end	0.000014
query end	0.000018
closing tables	0.000061
freeing items	0.000050
logging slow query	0.000002
cleaning up	0.000040

图中标红部分显示了两次优化器的执行时间差异。

小结

1)根据机器配置估算一个可接受的时间，用于优化器选择join顺序。

2)用profiling确定是否设置了过大的optimizer_search_depth。

3)业务上优化，尽量不要使用超过10张表的多表join。

4)PS:不要相信银弹。MySQL文档说设置为0则表示能够自动选择optimizer_search_depth的合理值，实际上代码上策略就是，如果join表数N<=7，则optimizer_search_depth=N+1，否则选N.

原文链接:http://dinglin.iteye.com/blog/2069124

阿里云OCS超时问题的分析与解决

作者: 阿里云技术团队

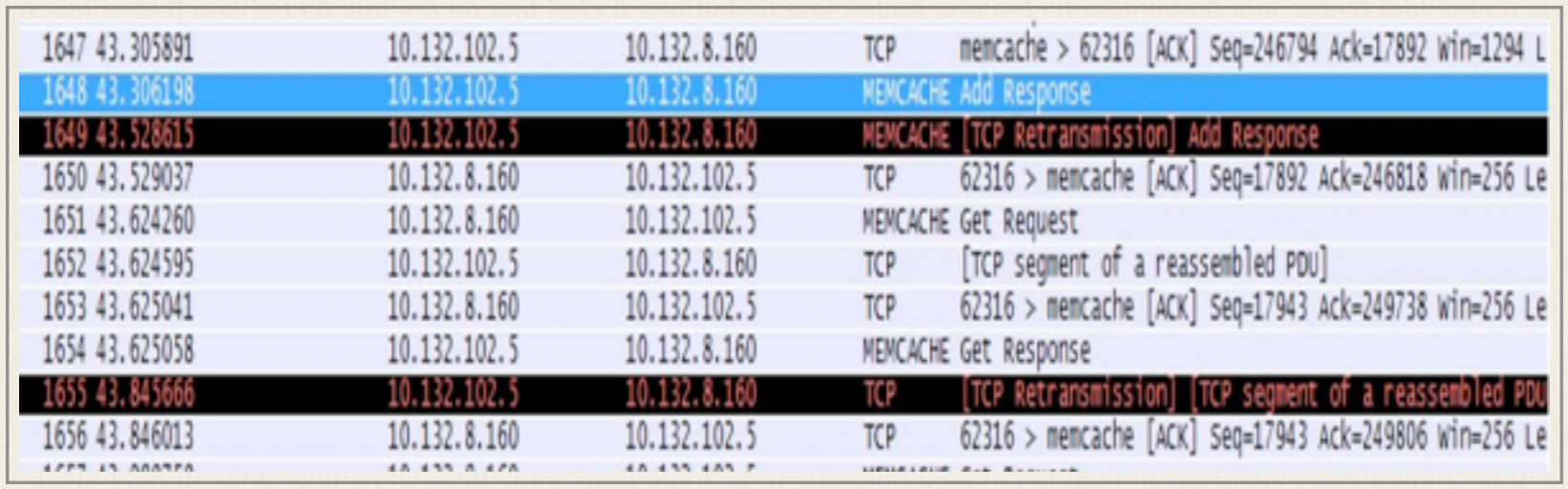
之前有用户联系我们阿里云，反映在使用OCS时会出现超时错误，希望我们阿里云技术团队能够帮忙解决。通常用户会将热点数据存放到OCS中，用以提高用户的业务处理响应速度，因此超时问题对于OCS来讲非常敏感，这引起了阿里云OCS团队的重视，随即开始了调查分析。

对于一个网络服务，通常导致超时的原因包括：网络抖动、CPU某个核心负载过高、内存不足导致的频繁swap、网卡负载过高、协议BUG导致的回包有误。通过分析，我们发现了客户端自身存在的某些问题导致OCS超时；除此之外，进一步的分析表明在某些特殊情况下OCS自身的一些问题也会导致超时。下面我们来看看阿里云OCS团队的工程师们是怎么样分析并解决这些超时问题的。

在进入技术细节分析之前，先简单介绍一下阿里云OCS的大致工作机制：基于Memcached协议的用户请求从阿里云ECS服务器上发出，通过阿里云SLB（负载均衡）连到阿里云OCS的前端proxy，再连接后端底层的服务器集群进行最终处理。

对于超时问题，我们最先考虑问题可能在于客户端到SLB，SLB到proxy之间的网络及协议问题。为了直观分析排查问题，在考虑proxy承受能力之后，我们建议用户不经过阿里云SLB直接连我们的proxy，这样可以直接在服务器抓包了解异常情况时的报文交互，同时也排除了SLB可能出问题的干扰。

在用户将自己的服务器切到我们的proxy上后，我们对每个客户端抓包，其中一台的报文如下：

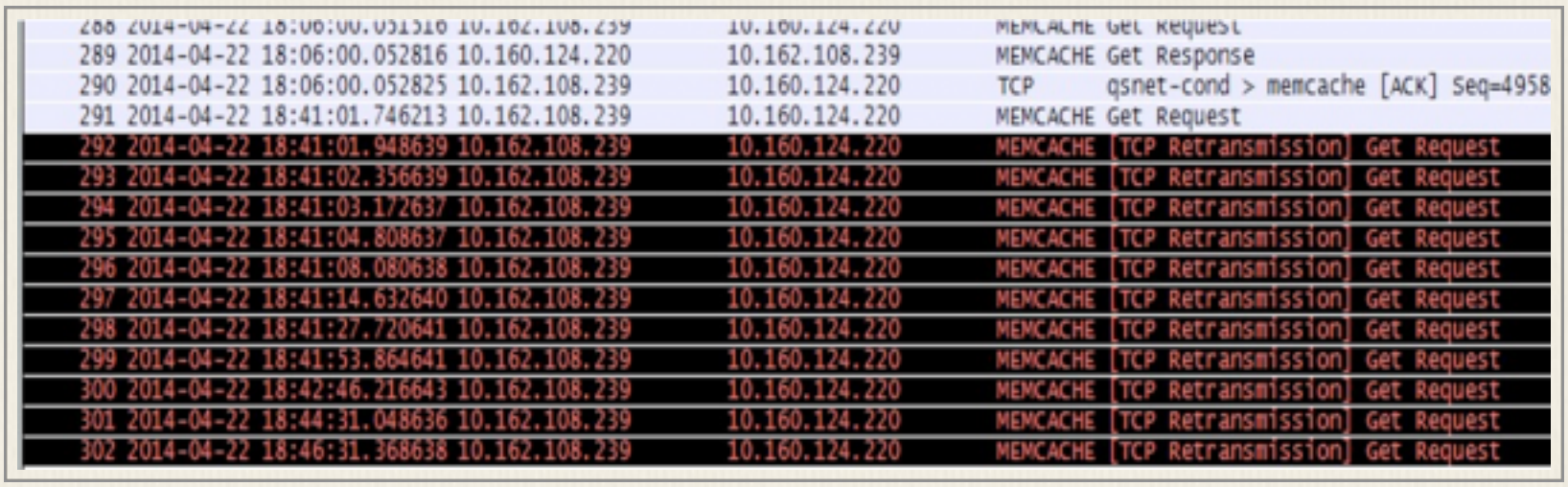


我们看到截图片段中有两个客户端给proxy的回报，超过了200ms，这说明客户端负载有问题，经过与用户确认这一情况得到了证实。由此我们知道：某些情况下使用OCS出现超时是用户客户端负载问题导致，只要合理配置客户端就能解决此类问题。

接下来我们进一步调查发现，另外还有一些超时是因为阿里云OCS自身的原因导致的。阿里云OCS对于存储数据的大小是有限制的，即Value最大为1MB。若超出此限制，客户端会收到“Value too large”的错误信息。某些OCS超时错误恰恰是这个尺寸大小导致的。上文提到OCS的处理流程是proxy调用底层后端服务，我们分析OCS代码发现底层后端代码中对报文Value长度的限制是 1000 * 1000字节，而proxy层根据memcached协议是1024 * 1024字节。这种不一致导致当客户端发送的Value长度处于1000 * 1000 ~ 1024 * 1024字节的临界区间时，set\replace\add操作没有任何信息返回，即没有response；而客户端会一直等待这个response，直到抛出超时异常。虽然是个看起来很初级的BUG，但由于它仅在某些特殊情况下出现，所以定位它还是花了一点时间。找到了这个问题，解决起来就容易了。

解决了上述客户端负载和尺寸检测BUG的问题，仍有另外的用户给我们报告OCS超时问题。莫非还有其他的原因?

对于新出现的这些超时情况，我们还是采取在客户端抓包的办法，看到的数据如下:



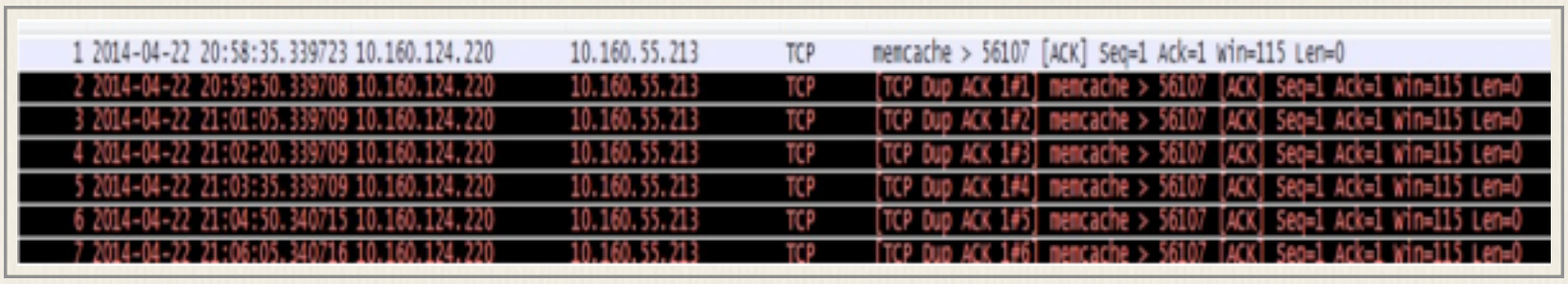
288	2014-04-22 18:06:00.051316	10.162.108.239	10.160.124.220	MEMCACHE Get Request
289	2014-04-22 18:06:00.052816	10.160.124.220	10.162.108.239	MEMCACHE Get Response
290	2014-04-22 18:06:00.052825	10.162.108.239	10.160.124.220	TCP qsnet-cond > memcache [ACK] Seq=4958
291	2014-04-22 18:41:01.746213	10.162.108.239	10.160.124.220	MEMCACHE Get Request
292	2014-04-22 18:41:01.948639	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
293	2014-04-22 18:41:02.356639	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
294	2014-04-22 18:41:03.172637	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
295	2014-04-22 18:41:04.808637	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
296	2014-04-22 18:41:08.080638	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
297	2014-04-22 18:41:14.632640	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
298	2014-04-22 18:41:27.720641	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
299	2014-04-22 18:41:53.864641	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
300	2014-04-22 18:42:46.216643	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
301	2014-04-22 18:44:31.048636	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request
302	2014-04-22 18:46:31.368638	10.162.108.239	10.160.124.220	MEMCACHE [TCP Retransmission] Get Request

从图中可以得知，客户端10.162.108.239到OCS服务器端10.160.124.220的连接在18:06分开始休眠到18:41被唤醒，发起一次Get请求；之后客户端经历多次重试后，TCP才知道连接已断。这期间客户端未从服务器端收到任何报文。

上文提到过，用户的请求要通过阿里云SLB（负载均衡）连到阿里云OCS的多台proxy，再连接后端底层的服务器集群。所以我们考虑是否存在SLB导致超时问题的可能性。

经过与阿里云SLB部门的同学沟通得知，SLB会断开15分钟未活跃的连接，且不会给客户端和服务器发任何FIN或RST报文。这样就可能出现一个问题，即客户端和服务器都认为连接正常。然而当客户端应用层发起一个请求时，该请求被交至TCP层，而TCP层不知道此时链接已经断开，于是重发该请求数次，直到应用层的超时时间到，就返回Timeout。在某些极端情况下，如果应用层没有设过期或者过期时间非常长，就会一直等到TCP层超时才会返回。

在得知了SLB的这个情况后，为了绕开这个问题，我们在OCS服务器上设置了TCP层的keepAlive包。将/proc/sys/net/ipv4/tcp_keepalive_time由默认的7200s改为450s一次。接着我们做测试，客户端连接到OCS服务器上，然后抓包如下：



图中显示在20:58分时，服务端TCP层触发keepalive包，但始终不能发出去。即客户端依然收不到任何信息。在客户端同时抓包，抓不到任何来自服务器11211端口的KeepAlive报文。接下来我们查看客户端的netstat状态如下：

即客户端依然认为自己还处于连接状态，因为没有收到服务器的任何消息。如果此时用户发起请求，仍然会得到最初用户反馈的现象，即继续重传直至超时。

看来这个问题还没有解决。我们继续分析，考虑到一个可能性：我们阿里云OCS的客户端都是跑在阿里云ECS上，会不会这里有什么情况?

于是我们与阿里云ECS团队的同学沟通了解到：ECS上开启了netfilter，其中一条过滤规则是对于空闲时间大于180s的连接，netfilter会将它从established表中剔除，且不会通知客户端服务器，当客户端或者服务端还在该链接上面发送报文时，NODE将不再转发。

至此情况大致明朗了：因为是通过ECS和SLB建立的连接，而ECS和SLB对于连接空闲时间都有各自不同的限制，所以可能出现OCS服务器与客户端之间的KeepAlive包丢失，从而导致超时情况的发生。于是我们把OCS服务器的KeepAlive改为90s，绕过SLB、ECS，启动应用再重新测试，问题解决。

表面上看起来同样都是OCS超时的问题，其真正的原因却是各不相同。我们在这里描述起来看似很轻松，但是在实际工作中能够迅速及时的解决这些问题却绝非易事。有的很容易就被发现解决，有的原因找起来却是破费脑筋。通过我们阿里云工程师们的努力，能够让用户更好的使用阿里云服务，这是我们工作最大的乐趣和价值所在。

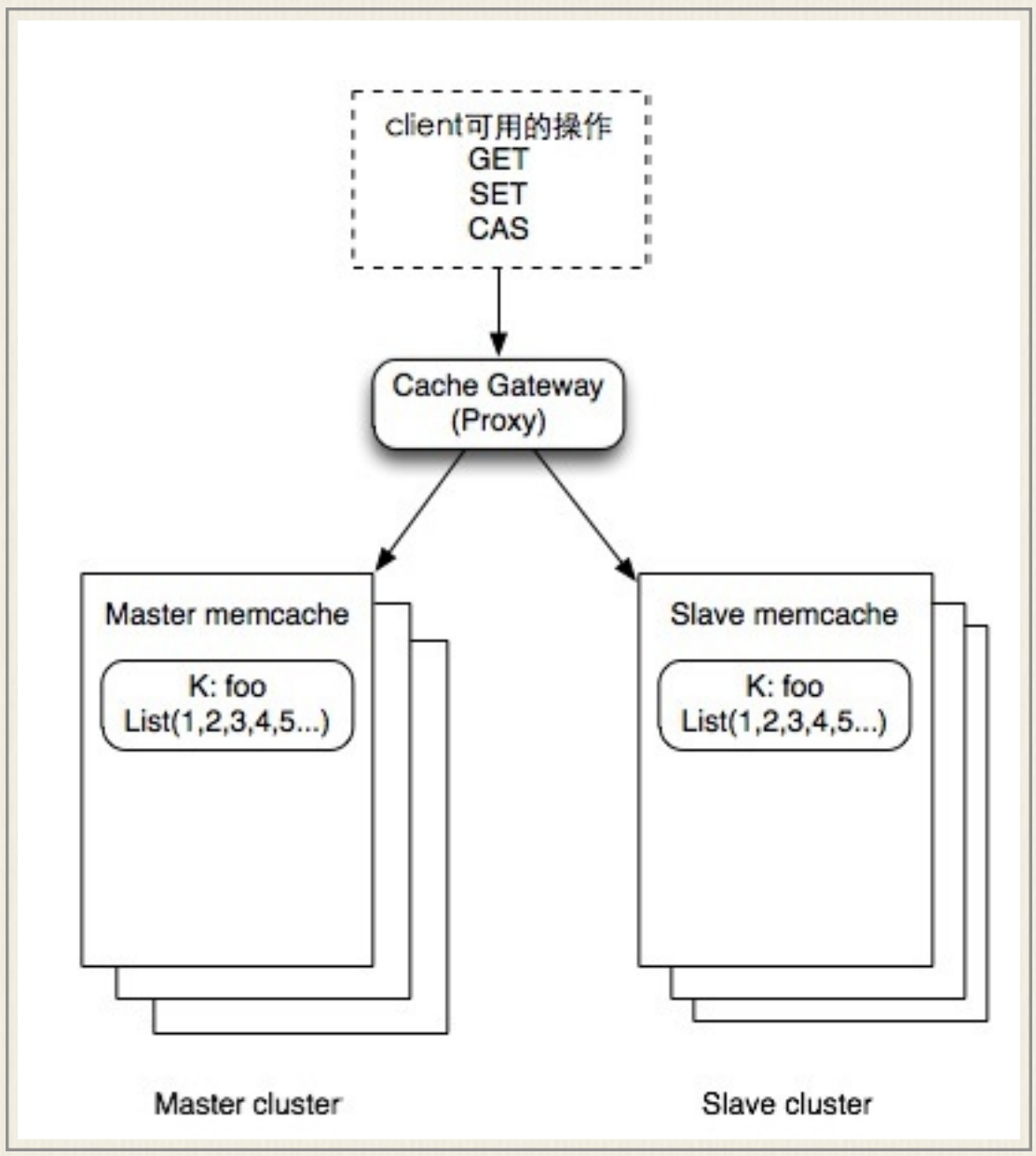
原文链接:http://blog.aliyun.com/341

分布式缓存的一起问题

作者: Tim

背景说明

分布式缓存中为了可用性及高性能的考虑，可以使用如下一种master/slave设计模式。



图中的proxy是逻辑的概念，可以是基于client的包装实现，也可以是独立的proxy服务，但本文大部分是指独立的服务。几个主要的问题说明如下。

为什么cache要使用两个集群((master/slave)来存放?

主要出于可用性及高性能的考虑。传统的架构使用基于一致性哈希的分布式缓存，数据只存在一份副本，在出现cache节点单点故障时，虽然可以由一致性哈希算法将请求均匀落到其他节点，但由于穿透的请求较多，仍然给数据库带来较大的访问压力。为了避免对数据穿透带来的冲击，数据使用两份副本可以避免穿透的问题。同时在数据访问较大时候，也可以更好的分担流量，避免峰值单份数据跑满对系统带来的冲击。

为什么两份副本要使用master/slave结构?

由于大型系统中通常存在多个client同时操作同一份数据，需要确保所有client对数据修改时数据的一致性。为了避免两cluster两份副本数据不一致带来的困扰，使用了一个简单的做法，在配置中人为指定一个cluster为master，所有的数据以master为准。

为什么一些场景需要使用CAS?

CAS在计算机并发领域通常指Compare-and-swap，在memcached中，也称为Check And Set. 在分布式系统中，一份数据可能同时被多个调用修改，比如微博中的@箱，一个用户同时收到多个@的情况还是比较常见，比如当原来@箱里面记录是{1，2，3}时，4和5由不同的调用来源同时到达，如果没有同步的保护，系统的数据有可能最终被写成{1,2,3,4}或{1,2,3,5}，由于memcached没有原生的list结构，list都是一个自定义的value, 则很容易出现client A覆盖了同时在写的client B的数据。因此假如两个调用方同时读到{1,2,3}时，第一个写入{1,2,3,4}会成功，后续的{1,2,3,5}CAS写入就会失败，因为此时服务器已经不是{1,2,3}了，失败的调用向服务端取回{1,2,3,4}，最终写入{1,2,3,4,5}

在master/slave场景，比起普通的memcache CAS有什么区别?

目前的做法是master cas成功之后，直接修改slave，并不同时在slave执行cas操作。由于数据存在两份副本，当数据不一致时，无法自动处理数据的不一致冲突。因此在实践上只以master操作为准。

为什么使用proxy?

使用proxy主要是出于可用性、命中率以及可运维方面的考虑

可用性与可运维：当进行服务器增容或缩容时，如果client的数量较大，如果未使用proxy模式，client所在服务器通常需要修改配置并且逐个重启。重启（系统维护）一方面带来可用性方面的问题，运维方面也较为繁琐。

命中率：如果业务场景需要较高的命中率（比如>90%），则增容或缩容就变得较为复杂，需要client配合做一些策略，比如扩容后仍然访问扩容前旧的节点的数据以保证命中率。如果用proxy模式则极大降低client的访问复杂性，将相关逻辑都封装在proxy之后。

分布式缓存的一起问题

最近某业务有一起master单点故障，导致在问题的时间段内，用户看不到最近发生变更的数据。由于在上述场景中，实现cas时候的流程如下

1) master.cas(k,v)
2) 如果1成功，slave.set(k,v)
3) 如果1失败，不执行slave.set()，直接return;

由于第三步在失败时，并不会set slave，导致数据出现一致性问题，即使slave依然可用，新的数据不会写入cache。

首先看在master failure时，为什么不切换到slave cas?

先说自动切换的问题

上文也提过，两份数据副本在出现数据不一致后，并不能自动仲裁达到最终一致性，但是指定master角色可以达到最终一致性。如果master角色可以由调用方自动切换，则会带来数据的混乱。调用方存在多个节点，至少需要统一的config server来保证切换的一致性。另外，自动切换发生后，无法达到两份数据的最终一致性。

再说由运维手工切换

由于不牵涉到代码的逻辑判断，虽然切换也会带来一些数据一致性问题，在具体场景下（比如master长久宕机）切换可以接受。

在出现上述问题后，其他一些解决方案如下。

1. proxy在master cas失败时候delete slave data
2. client在master cas失败时set slave, 并且将数据过期时间设成5分钟

上述方案很难完美，一些明显存在的问题如下

方案1：

命中率的问题。由于delete导致修改的数据迅速失效，会导致读取量的增加，在读写均密集的业务场景，可能会导致数据访问出现波动。

接口职责单一性的问题。proxy在cas调用中隐藏了删除数据的逻辑，这是一个未在正常期望范围内的额外操作，在特殊情况下，可能会导致不可预料的情况出现。(尽管在实际操作中proxy提供配置开关选项)

方案2：

依然是命中率的问题，5分钟过期延缓了过期的访问数据库的压力，但相关压力仍然会传递到数据库。

希望通过上面说明读者能理解这个场景的问题。在这个场景下，完美的方案应当如何设计?

原文链接:http://timyang.net/data/cache-failure/

Query意图分析：记一次完整的机器学习过程（scikit learn library学

作者: zero_learner

所谓学习问题，是指观察由n个样本组成的集合，并根据这些数据来预测未知数据的性质。

学习任务（一个二分类问题）：

区分一个普通的互联网检索Query是否具有某个垂直领域的意图。假设现在有一个O2O领域的垂直搜索引擎，专门为用户提供团购、优惠券的检索；同时存在一个通用的搜索引擎，比如百度，通用搜索引擎希望能够识别出一个Query是否具有O2O检索意图，如果有则调用O2O垂直搜索引擎，获取结果作为通用搜索引擎的结果补充。

我们的目的是学习出一个分类器（classifier），分类器可以理解为一个函数，其输入为一个Query，输出为0（表示该Query不具有o2o意图）或1（表示该Query具有o2o意图）。

特征提取：

要完成这样一个学习任务，首先我们必须找出决定一个Query是否具有O2O意图的影响因素，这些影响因素称之为特征（feature）。特征的好坏很大程度上决定了分类器的效果。在机器学习领域我们都知道特征比模型（学习算法）更重要。（顺便说一下，工业界的人都是这么认为的，学术界的人可能不以为然，他们整天捣鼓算法，发出来的文章大部分都没法在实际中应用。）举个例子，如果我们的特征选得很好，可能我们用简单的规则就能判断出最终的结果，甚至不需要模型。比如，要判断一个人是男还是女（人类当然很好判断，一看就知道，这里我们假设由计算机来完成这个任务，计算机有很多传感器（摄像头、体重器等等）可以采集到各种数据），我们可以找到很多特征：身高、体重、皮肤颜色、头发长度等等。因为根据

统计我们知道男人一般比女人重，比女人高，皮肤比女人黑，头发比女人短；所以这些特征都有一定的区分度，但是总有反例存在。我们用最好的算法可能准确率也达不到100%。假设计算机还能够读取人的身份证号码，那么我们可能获得一个更强的特征：身份证号码的倒数第二位是否是偶数。根据身份证编码规则，我们知道男性的身份证号码的倒数第二位是奇数，女生是偶数。因此，有了这个特征其他的特征都不需要了，而且我们的分类器也很简单，不需要复杂的算法。

言归正传，对于O2O Query意图识别这一学习任务，我们可以用的特征可能有：Query在垂直引擎里能够检索到的结果数量、Query在垂直引擎里能够检索到的结果的类目困惑度（perplexity）（检索结果的类目越集中说明其意图越强）、Query能否预测到特征的O2O商品类目、Query是否包含O2O产品词或品牌词、Query在垂直引擎的历史展现次数（PV)和点击率(ctr)、Query在垂直引擎的检索结果相关性等等。

特征表示：

特征表示是对特征提取结果的再加工，目的是增强特征的表示能力，防止模型（分类器）过于复杂和学习困难。比如对连续的特征值进行离散化，就是一种常用的方法。这里我们以“Query在垂直引擎里能够检索到的结果数量”这一特征为例，简要介绍一下特征值分段的过程。首先，分析一下这一维特征的分布情况，我们对这一维特征值的最小值、最大值、平均值、方差、中位数、三分位数、四分位数、某些特定值（比如零值）所占比例等等都要有一个大致的了解。获取这些值，python编程语言的numpy模块有很多现成的函数可以调用。最好的办法就是可视化，借助python的matplotlib工具我们可以很容易地划出数据分布的直方图，从而判断出我们应该对特征值划多少个区间，每个区间的范围是怎样的。比如说我们要对“结果数量”这一维特征值除了“0”以为的其他值均匀地分为10个区间，即每个区间内的样本数大致相同。“0”是一个特殊的值，因此我们想把它分到一个单独的区间，这样我们一共有11个区间。python代码实现如下：

```
import numpy as np
```

```
def bin(bins):
    assert isinstance(bins, (list, tuple))
    def scatter(x):
        if x == 0: return 0
        for i in range(len(bins)):
            if x <= bins[i]: return i + 1
        return len(bins)
    return np.frompyfunc(scatter, 1, 1)

data = np.loadtxt("D:\query_features.xls", dtype='int')
# descrete
o2o_result_num = data[:,0]
o2o_has_result = o2o_result_num[o2o_result_num > 0]
bins = [ np.percentile(o2o_has_result, x) for x in range(10, 101, 10) ]
data[:,0] = bin(bins)(o2o_result_num)
```

我们首先获取每个区间的起止范围，即分别算法特征向量的10个百分位数，并依此为基础算出新的特征值（通过bin函数，一个numpy的universal function）。

训练数据：

这里我们通过有监督学习的方法来拟合分类器模型。所谓有监督学习是指通过提供一批带有标注（学习的目标）的数据（称之为训练样本），学习器通过分析数据的规律尝试拟合出这些数据和学习目标间的函数，使得定义在训练集上的总体误差尽可能的小，从而利用学得的函数来预测未知数据的

学习方法。注意这不是一个严格的定义，而是我根据自己的理解简化出来的。

一批带有标注的训练数据从何而来，一般而言都需要人工标注。我们从搜索引擎的日志里随机采集一批Query，并且保证这批Query能够覆盖到每维特征的每个取值（从这里也可以看出为什么要做特征分区间或离散化了，因为如不这样做我们就不能保证能够覆盖到每维特征的每个取值）。然后，通过人肉的方法给这边Query打上是否具有O2O意图的标签。数据标注是一个痛苦而漫长的过程，需要具有一定领域知识的人来干这样的活。标注质量的好坏很有可能会影响到学习到的模型（这里指分类器）在未知Query上判别效果的好坏。即正确的老师更可能教出正确的学生，反之，错误的老师教坏学生的可能性越大。在我自己标注数据的过程中，发现有一些Query的O2O意图比较模棱两可，导致我后来回头看的时候总觉得自己标得不对，反反复复修改了好几次。

选择模型：

在我们的问题中，模型就是要学习的分类器。有监督学习的分类器有很多，比如决策树、随机森林、逻辑回归、梯度提升、SVM等等。如何为我们的分类问题选择合适的机器学习算法呢？当然，如果我们真正关心准确率，那么最佳方法是测试各种不同的算法（同时还要确保对每个算法测试不同参数），然后通过交叉验证选择最好的一个。但是，如果你只是为你的问题寻找一个“足够好”的算法，或者一个起点，也是有一些还不错的一般准则的，比如如果训练集很小，那么高偏差/低方差分类器（如朴素贝叶斯分类器）要优于低偏差/高方差分类器（如k近邻分类器），因为后者容易过拟合。然而，随着训练集的增大，低偏差/高方差分类器将开始胜出（它们具有较低的渐近误差），因为高偏差分类器不足以提供准确的模型。

这里我们重点介绍一次完整的机器学习全过程，所以不花大篇幅在模型选择的问题上，推荐大家读一些这篇文章：《如何选择机器学习分类器？》。

通过交叉验证拟合模型：

机器学习会学习数据集的某些属性，并运用于新数据。这就是为什么习惯上会把数据分为两个集合，由此来评价算法的优劣。这两个集合，一个叫做训练集（train data），我们从中获得数据的性质；一个叫做测试集(test

data)，我们在此测试这些性质，即模型的准确率。将一个算法作用于一个原始数据，我们不可能只做出随机的划分一次train和test data，然后得到一个准确率，就作为衡量这个算法好坏的标准。因为这样存在偶然性。我们必须好多次的随机的划分train data和test data，分别在其上面算出各自的准确率。这样就有一组准确率数据，根据这一组数据，就可以较好的准确的衡量算法的好坏。交叉验证就是一种在数据量有限的情况下的非常好evaluate performance的方法。

```
from sklearn import cross_validation
from sklearn import tree
from sklearn import ensemble
from sklearn import linear_model
from sklearn import svm

lr = linear_model.LogisticRegression()
lr_scores = cross_validation.cross_val_score(lr, train_data, train_target, cv=5)
print("logistic regression accuracy:")
print(lr_scores)

clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=8, min_samples_split=5)
clf_scores = cross_validation.cross_val_score(clf, train_data, train_target, cv=5)
print("decision tree accuracy:")
print(clf_scores)

rfc = ensemble.RandomForestClassifier(criterion='entropy', n_estimators=3, max_features=
0.5, min_samples_split=5)
rfc_scores = cross_validation.cross_val_score(rfc, train_data, train_target, cv=5)
print("random forest accuracy:")
print(rfc_scores)

etc = ensemble.ExtraTreesClassifier(criterion='entropy', n_estimators=3, max_features=0.
6, min_samples_split=5)
etc_scores = cross_validation.cross_val_score(etc, train_data, train_target, cv=5)
print("extra trees accuracy:")
print(etc_scores)

gbc = ensemble.GradientBoostingClassifier()
gbc_scores = cross_validation.cross_val_score(gbc, train_data, train_target, cv=5)
print("gradient boosting accuracy:")
print(gbc_scores)

svc = svm.SVC()
svc_scores = cross_validation.cross_val_score(svc, train_data, train_target, cv=5)
print("svm classifier accuracy:")
print(svc_scores)
```

上面的代码我们尝试同交叉验证的方法对比五种不同模型的准确率，结果如下：

```
logistic regression accuracy:
[ 0.76953125  0.83921569  0.85433071  0.81102362  0.83858268]
decision tree accuracy:
[ 0.73828125  0.8         0.77559055  0.71653543  0.83464567]
random forest accuracy:
[ 0.75        0.76862745  0.76377953  0.77165354  0.80314961]
extra trees accuracy:
[ 0.734375    0.78039216  0.7992126   0.76377953  0.79527559]
gradient boosting accuracy:
[ 0.7578125   0.81960784  0.83464567  0.80708661  0.84251969]
svm classifier accuracy:
[ 0.703125    0.78431373  0.77952756  0.77952756  0.80708661]
```

在O2O意图识别这个学习问题上，逻辑回归分类器具有最好的准确率，其次是梯度提升分类器；决策树和随机森林在我们的测试结果中并没有体现出明显的差异，可能是我们的特殊数量太少并且样本数也较少的原因；另外大名典典的SVM的表现却比较让人失望。总体而言，准确率只有82%左右，分析其原因，一方面我们实现的特征数量较少；另一方面暂时未能实现区分能力强的特征。后续会对此持续优化。

由于逻辑回归分类器具有最好的性能，我们决定用全部是可能训练数据来拟合之。

lr = lr.fit(train_data, train_target)

模型数据持久化：

学到的模型要能够在将来利用起来，就必须把模型保存下来，以便下次使用。同时，数据离散化或数据分区的范围数据也要保存下来，在预测的时候同样也需要对特征进行区间划分。python提供了pickle模块用来序列号对象，并保存到硬盘上。同时，scikit-learn库也提供了更加高效的模型持久化模块，可以直接使用。

```
from sklearn.externals import joblib
joblib.dump(lr, 'D:\lr.model')
import pickle
bin_file = open(r'D:\result_bin.data', 'wb')
pickle.dump(bins, bin_file)
bin_file.close()
```

分类器的使用：

现在大功告成了，我们需要做的就是用学习到的分类器来判断一个新的Query到底是否具有O2O意图。因为我们分类器的输入是Query的特征向量，而不是Query本身，因此我们需要实现提取好Query的特征。假设我们已经离线算好了每个Query的特征，现在使用的时候只需要将其加载进内场即可。分类器使用的过程首先是从硬盘读取模型数据和Query特征，然后调用模型对Query进行预测，输出结果。

```
# load result bin data and model
bin_file = open(r'D:\result_bin.data', 'rb')
bins = pickle.load(bin_file)
bin_file.close()

lr = joblib.load('D:\lr.model')

# load data
query = np.genfromtxt(r'D:\o2o_query_rec\all_query', dtype='U2', comments=None, converte
rs={0: lambda x: str(x, 'utf-8')})
feature = np.loadtxt(r'D:\o2o_query_rec\all_features', dtype='int', delimiter='\001')

# descrite
feature[:,0] = bin(bins)(feature[:,0])
feature[:,1] = ufunc_segment(feature[:,1])

# predict
result = lr.predict(feature)

# save result
#np.savetxt(r'D:\o2o_query_rec\classify_result.txt', np.c_[query, result], fmt=u"%s", de
limiter="\t")
result_file = open(r'D:\o2o_query_rec\classify_result.txt', 'w')
i = 0
for q in query:
    result_file.write('%s\t%d\n' % (q, result[i]))
    i += 1
result_file.close()
```

需要注意的是我们Query的编码是UTF-8，load的时候需要做相应的转换。另外，在python 3.3版本，numpy的savetxt函数并不能正确保持UTF-8格式的中文Query（第20行注释掉的代码输出的Query都变成了bytes格式的），如果小伙伴们有更好的办法能够解决这个问题，请告诉我，谢谢！

原文链接:http://www.cnblogs.com/yangxudong/p/3750358.html

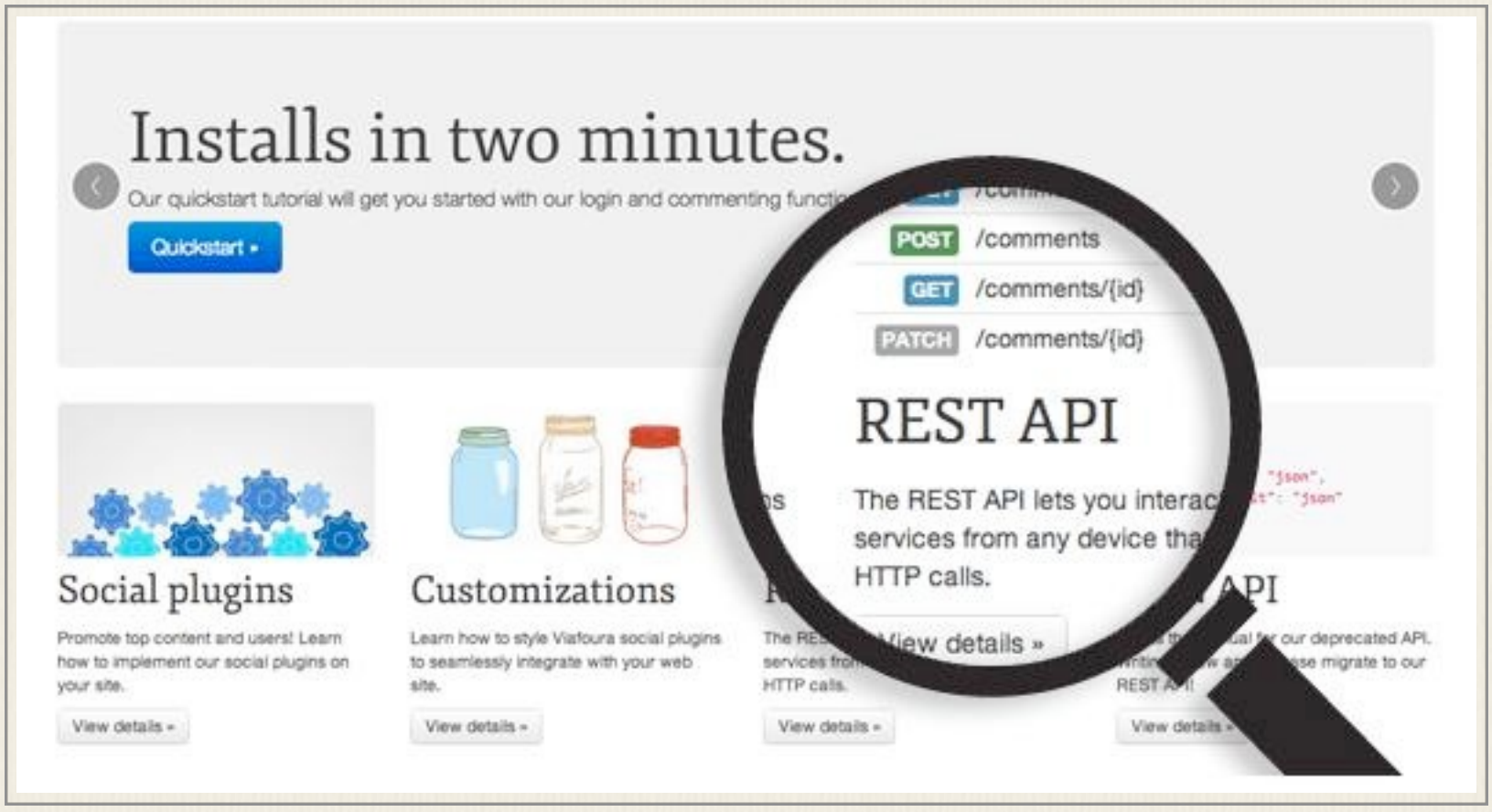
RESTful API 设计指南

作者: 阮一峰

网络应用程序，分为前端和后端两个部分。当前的发展趋势，就是前端设备层出不穷（手机、平板、桌面电脑、其他专用设备.....）。

因此，必须有一种统一的机制，方便不同的前端设备与后端进行通信。这导致API构架的流行，甚至出现"API First"的设计思想。RESTful API是目前比较成熟的一套互联网应用程序的API设计理论。我以前写过一篇《理解RESTful架构》，探讨如何理解这个概念。

今天，我将介绍RESTful API的设计细节，探讨如何设计一套合理、好用的API。我的主要参考资料是这篇《Principles of good RESTful API Design》。



一、协议

API与用户的通信协议，总是使用HTTPs协议。

二、域名

应该尽量将API部署在专用域名之下。

https://api.example.com

如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下。

https://example.org/api/

三、版本（Versioning）

应该将API的版本号放入URL。

https://api.example.com/v1/

另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观。

四、路径（Endpoint）

路径又称"终点"（endpoint），表示API的具体网址。

在RESTful架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以API中的名词也应该使用复数。

举例来说，有一个API提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

https://api.example.com/v1/zoos

https://api.example.com/v1/animals

https://api.example.com/v1/employees

五、HTTP动词

对于资源的具体操作类型，由HTTP动词表示。

常用的HTTP动词有下面五个（括号里是对应的SQL命令）。

GET（SELECT）：从服务器取出资源（一项或多项）。

POST（CREATE）：在服务器新建一个资源。

PUT（UPDATE ）：在服务器更新资源（客户端提供改变后的完整资源）。

PATCH（UPDATE ）：在服务器更新资源（客户端提供改变的属性）。

DELETE（DELETE）：从服务器删除资源。

还有两个不常用的HTTP动词。

HEAD：获取资源的元数据。

OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

下面是一些例子。

GET /zoos：列出所有动物园

POST /zoos：新建一个动物园

GET /zoos/ID：获取某个指定动物园的信息

PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）

PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）

DELETE /zoos/ID：删除某个动物园

GET /zoos/ID/animals：列出某个指定动物园的所有动物

DELETE /zoos/ID/animals/ID：删除某个指定动物园的指定动物

六、过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

下面是一些常见的参数。

?limit=10：指定返回记录的数量

?offset=10：指定返回记录的开始位置。

?sortby=name&order =asc：指定返回结果按照哪个属性排序，以及排序顺序。

?animal_type_id=1：指定筛选条件

参数的设计允许存在冗余，即允许API路径和URL参数偶尔有重复。比如，GET /zoo/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。

七、状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。

200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。

201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。

204 NO CONTENT - [DELETE]：用户删除数据成功。

400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。。

404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。

500 INTERNAL SERVER ERROR - [*]：服务器发生错误，用户将无法判断发出的请求是否成功。

状态码的完全列表参见这里。

八、错误处理（Error handling）

如果状态码是4xx，就应该向用户返回出错信息。一般来说，返回的信息中将error作为键名，出错信息作为键值即可。

```
{
    error: "Invalid API key"
}
```

九、返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

GET /collection：返回资源对象的列表（数组）

GET /collection/resource：返回单个资源对象

POST /collection：返回新生成的资源对象

PUT /collection/resource：返回完整的资源对象

PATCH /collection/resource：返回完整的资源对象

DELETE /collection/resource：返回一个空文档

十、Hypermedia API

RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向api.example.com的根目录发出请求，会得到这样一个文档。

```
{"link": {
  "rel":   "collection https://www.example.com/zoos",
  "href":  "https://api.example.com/zoos",
  "title": "List of zoos",
  "type":  "application/vnd.yourformat+json"
}}
```

上面代码表示，文档中有一个link属性，用户读取这个属性就知道下一步该调用什么API了。rel表示这个API与当前网址的关系（collection关系，并给出该collection的网址），href表示API的路径，title表示API的标题，type表示返回类型。

Hypermedia API的设计被称为HATEOAS。Github的API就是这种设计，访问api.github.com会得到一个所有可用API的网址列表。

```
{
  "current_user_url": "https://api.github.com/user",
  "authorizations_url": "https://api.github.com/authorizations",
  // ...
}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问api.github.com/user，然后就得到了下面结果。

```
{
  "message": "Requires authentication",
```

"documentation_url": "https://developer.github.com/v3"

}

上面代码表示，服务器给出了提示信息，以及文档的网址。

十一、其他

（1）API的身份认证应该使用OAuth 2.0框架。

（2）服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

（完）

原文链接:http://www.ruanyifeng.com/blog/2014/05/restful_api.html

从Google+更新说说Navigation Drawer

作者: Stephen Day

Oh, the new Google+ client made my mind blown.

近半年来， Google 在 Android 平台上的各种应用更新，虽然从UX-er的角度，对其目的能揣测个大概，但至少我没有再看到过像 “Navigation Drawer的诞生” 这类出自Android 部门员工口的详细修改意图，所以我也无法验证 Google 在 Play Music 中去掉 Overflow Menu， Google+ 中采用三个圆形 dots 来作为 Overflow Menu 图标，还有一些应用将 setting 和 help 移入到 Navigation Drawer 中去， 以及等等修改的目的。

昨天晚上，在用过更新后的 Google+ 客户端后，对我来说，最大的变化大概有两个

1. 去掉了 Navigation Drawer
2. Action Bar变成了56DP

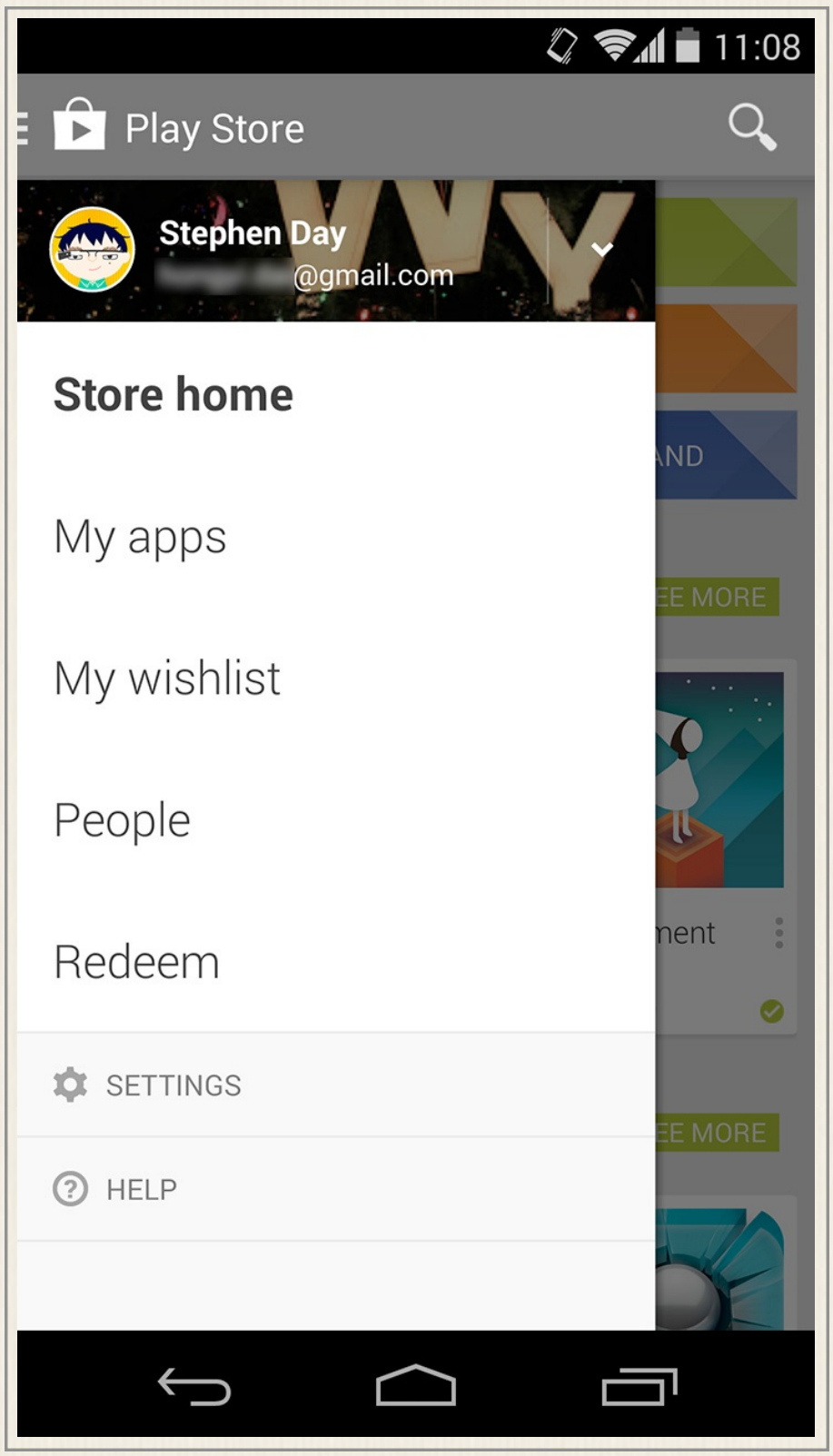
对于第一点从视觉上就会发现变化非常大了，但对于第二点，WTH?

总所周知，对于小屏幕移动设备 Actoin Bar 的高度是48DP所有应用都在采用的一个数值，但惟有 Google+ 这次将数值设置成了一个平板设备应该具备的 Action Bar 高度，令人匪夷所思。

重新回到第一点：去掉了 Navigation Drawer。

从去年 I/O 大会之后，为了控制 Android 平台日益混乱的抽屉交互方式， Google 将 Navigation Drawer 纳入了 Android Design 规范当中，随后

大量应用开始采用这种交互模式，然而也出现了一些质疑，但大多数质疑声都集中在这个 Drawer 的可发现性上。



也就是说“在 Drawer 未被呼出时，我怎么知道这里有个导航呢？”

当时在 Adam 的文章里是这样解释的：

The next issue to tackle was discoverability. Thanks to work already done by the developer community users were already accustomed to reading a three horizontal line icon at the left side of the action bar as a menu/

drawer button. (During user testing one user referred to it as “the hamburger” and the name stuck.) We knew this approach was a sure thing fori discoverability but it didn’t quite sit right. Replacing the app icon meant potentially losing a big part of the app’s identity.

下一个需要解决的问题就是可发现性。多谢那些开发者社群中的用户所做的工作，已经习惯将三条水平线的图标放在action bar左边作为一个菜单/抽屉按钮。（在用户测试过程中，一位用户将其认为是一个“汉堡包”。）我们知道这种途径无疑增强了可发现性，但有一点不对劲。替换应用图标意味着丢失大量应用特性。

The apps that made the full-size three-line button icon in the upper left popular are themselves popular apps with strong branding throughout their UI. For other apps, using that full-size hamburger icon in place of the app’s own icon on the action bar meant losing a glanceable identifying characteristic. It ran the risk of apps looking too “samey.” We decided to play around with changing the Up chevron to the left of the icon as an alternative to a full hamburger.

那些流行将整个三条线的按钮放在左上角的应用,他们可以通过自己的UI来表达强烈的品牌感。对于其他的应用，使用全尺寸汉堡包图标替换应用action bar上自己的图标意味着失去自己的识别特征。它会使众多应用看起来过于相似。我们决定将向上箭头替换为左移后的汉堡包图标，来代替整个图标。

A first draft of the small hamburger (affectionately called “the slider”) didn’t test well. Making it the same width as the Up chevron made it too easy for users to miss in a quick visual scan. Some of our visual design team got to work creating a number of alternatives for later review. In the meantime we added the long-edge touch “peek” to DrawerLayout’s behavior as a hint to a drawer’s presence, but we knew this wouldn’t be sufficient for discoverability on its own.

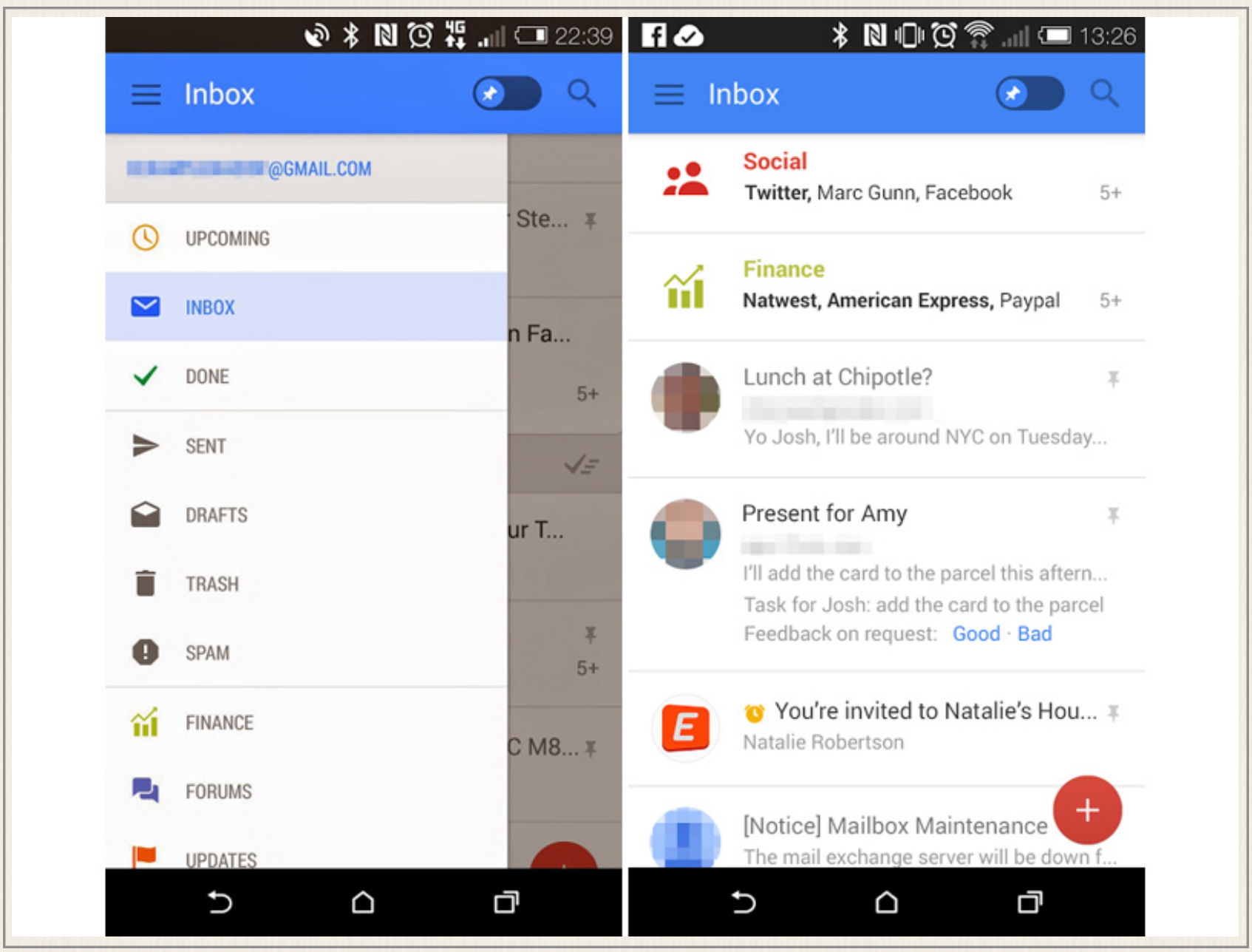
第一版小汉堡（亲切地称为“滑块”）的草图感觉不太好。如果将其宽度设置得跟向上图标一样，用户在快速扫视的过程中很容易忽视掉。我们视觉设计团队的成员又做了一些候选的版本。与此同时，我们将“触摸整条屏幕边

缘即可窥视抽屉”加入了抽屉式布局的交互行为当中，它可以作为一个抽屉存在的提示，但是我们知道这是不够的。

现在回味起来，第二段中的红色文字是否有些牵强，Android 在规范之初就拿自己的应用做了榜样：要在 Action Bar 左侧加入应用的 icon ，他们称之为 Branding ，但至少纵观 iOS 应用来看，Branding 并不一定需要一个 icon 放在 Bar上，并且在我个人的工作中，我用 Branding 这个理由未能说服让设计师或者 leader 同意在 Action Bar 上加一个应用 icon 。

如果 Action Bar 上不需要一个 icon ，按照上文中的逻辑， Navigation Drawer 就可以使用完全露出的小汉堡图标。

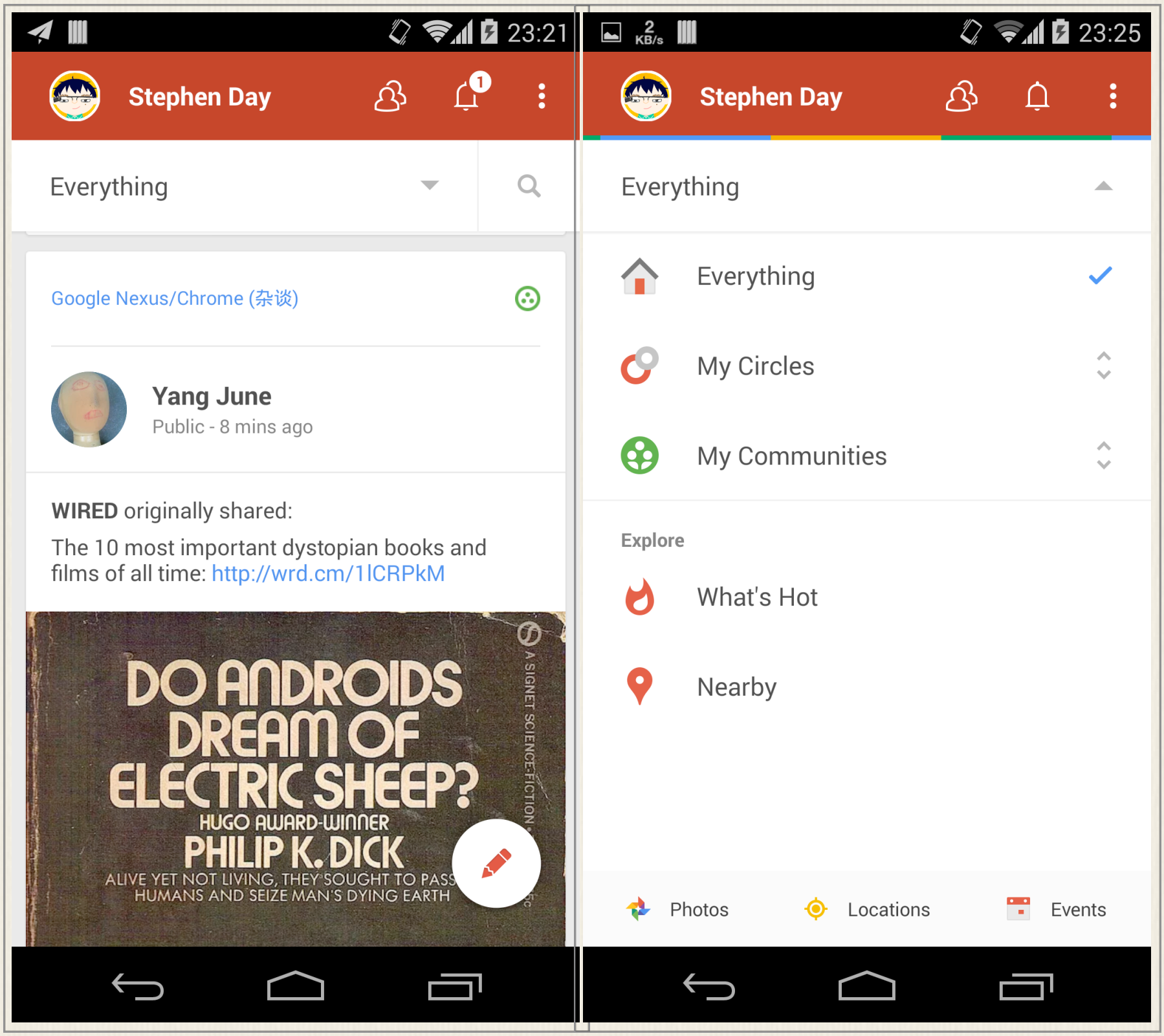
基于这样的推测，我不难理解在 Navigation Drawer 盛行的今天，竟还会出现以下“逆天”的“泄露图”：



（在最近一次 Gmail 更新中，在 UI 上做了少量的修改。）

回到正题，为什么这次 Google+ 去掉了 Navigation Drawer ，从生态系统多样性的角度，我不愿意所有的应用都采用一种导航方式，虽然这样大量节约了用户的学习成本，但所有的应用都 think the same 毕竟不是好事；可能原因仅仅是因为 Google 自己看厌了，又或者是临时工干的， Who knows， Google ， Who knows?

但至少这次改变至少把导航放到了显眼的位置。（但某些功能用 Android Police 的话来说变成了 super hidden -_-）

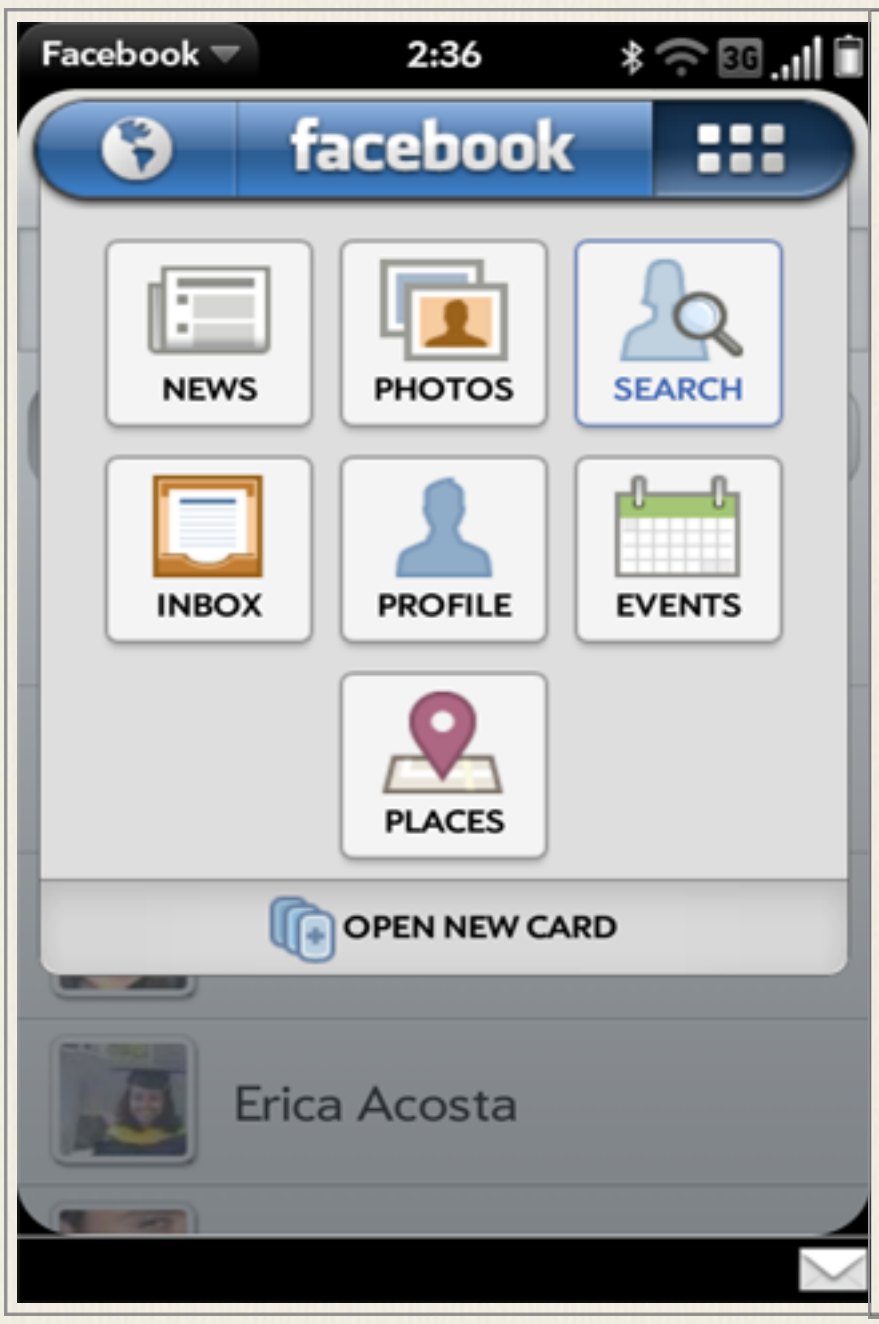


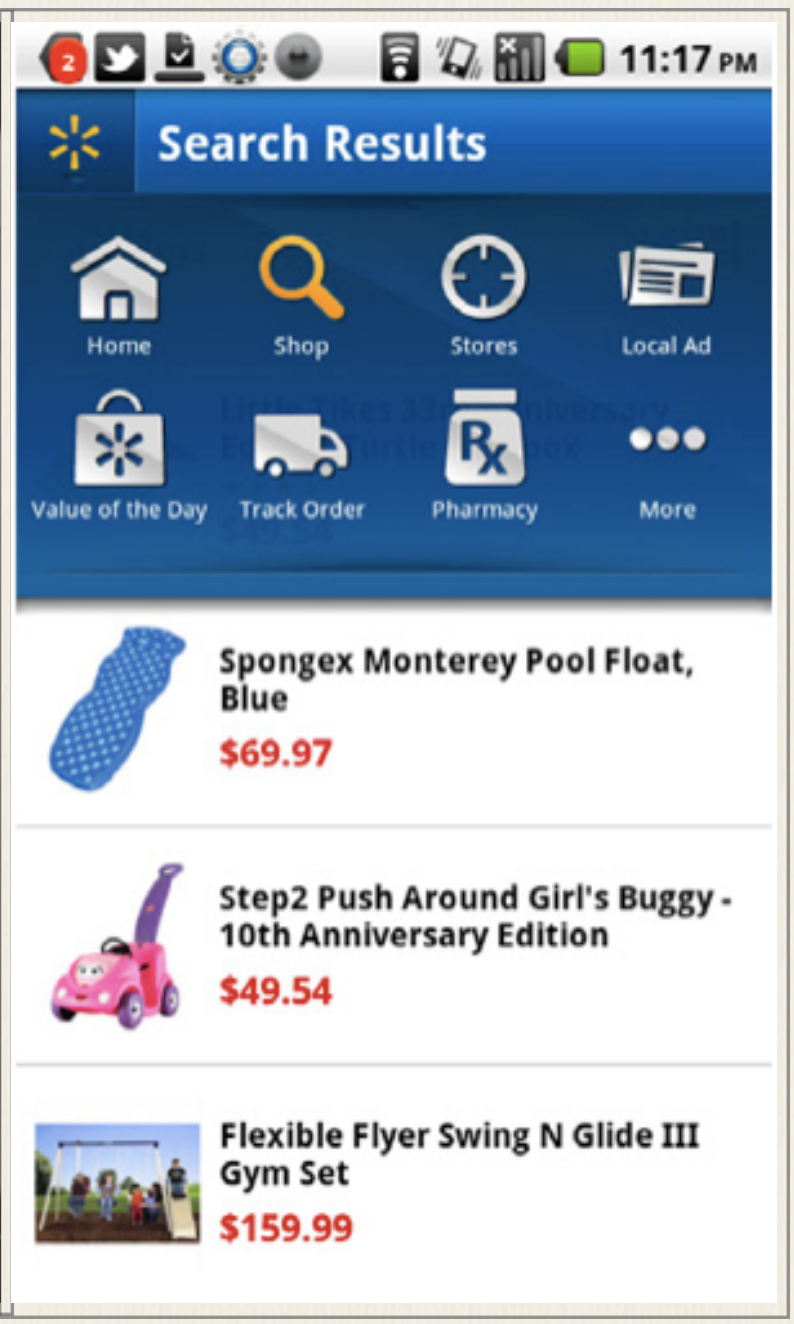
当我第一眼看到这个新的 Navigation 时，我第一感受是似曾相识，如果你看过 “Mobile Design Pattern Gallery” 这本书应该会发现书中提到了两种导航模式：

第一种叫做 “Mega Menu”，书中的解释是：

A mobile Mega Menu is like the web Mega Menu, a big overlay panel with custom formatting and grouping of the menu options. The RipCurl website uses a mega menu for navigating into sub categories of clothing.

举例是 webos 上的 Facebook 和 Android 上的 Walmart。

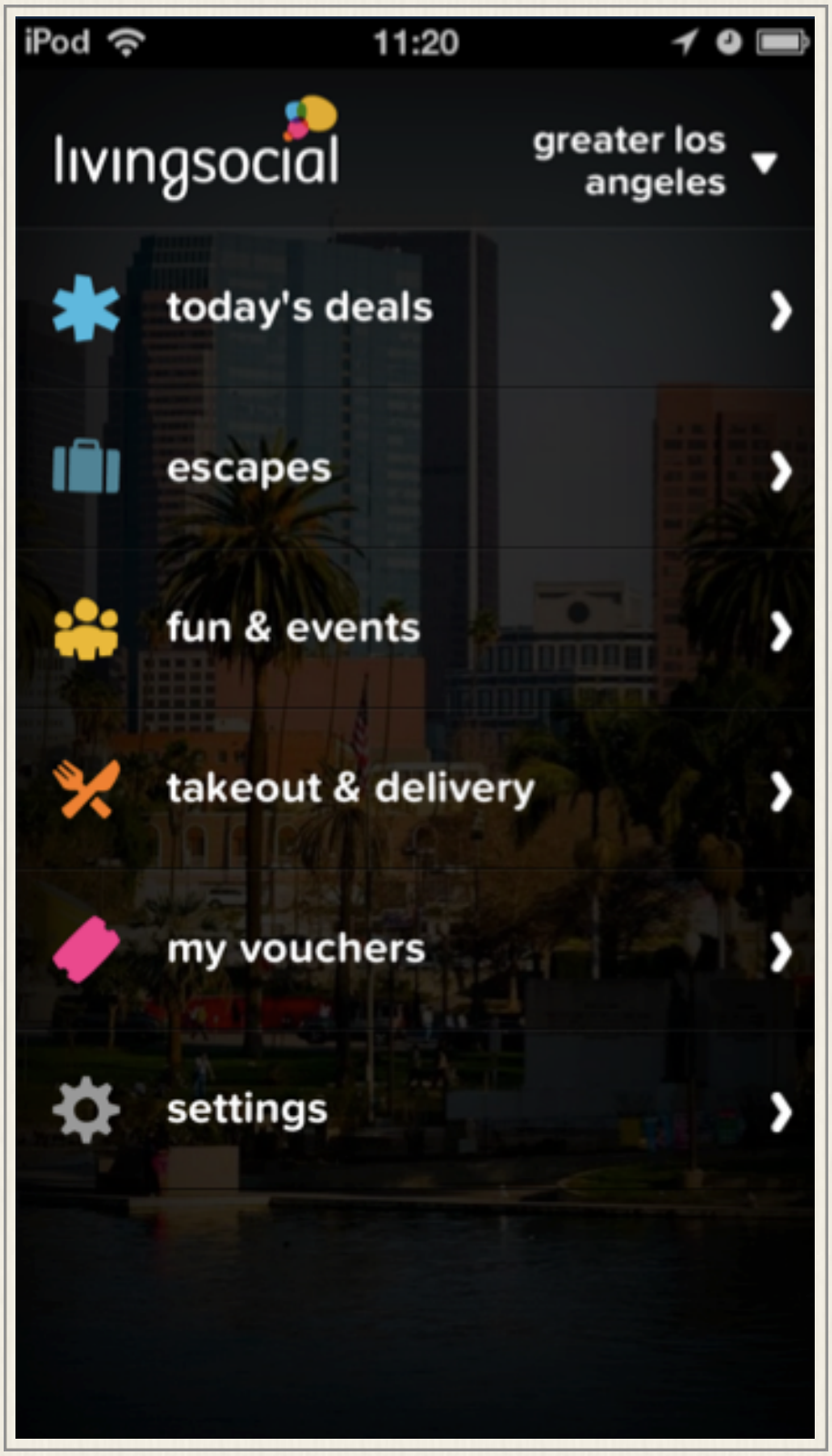




还有一种导航模式叫做 “List Menu”:

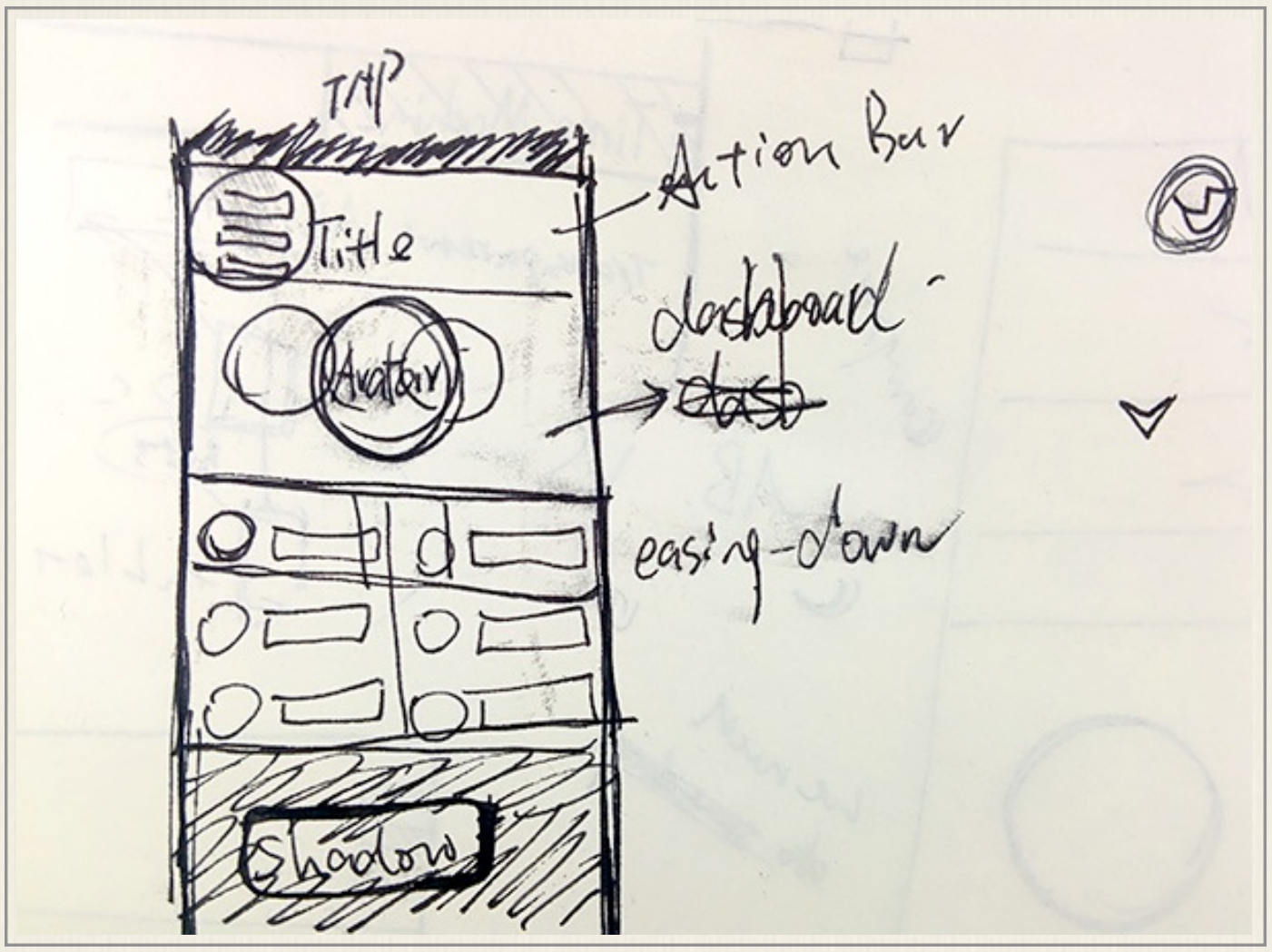
The List Menu is similar to that Springboard in that each is a jumping off point into the application. There are numerous variations of this pattern including personalized list menus, grouped lists, and enhanced lists. Enhanced lists are simple List Menus with additional features for searching, browsing or filtering.

举例很多，我找出两个相对新的应用， Livingsocial 和周边快查:



现在看来新版 Google+ 的新导航就是 Mega Menu + List Menu，比较让人奇怪的是 Mega Menu 已经很少在现有主流应用中找到了，而 Google 现在采用这样一个导航模式让人有些捉摸不透，Google 要放弃 Navigation Drawer？ 我觉得不至于做到放弃这一步，但也许就是时候改变一下了。

不过话说回来，之所以新版的 Google+ 让我这么兴奋，主要原因是还是要回到开头那个“似曾相识”，因为就在两周前，我对某个应用的 redesign 便尝试着去掉了 Navigation Drawer 而采用了 Mega Menu 的形式（想到一块去了？！），其实抽象看来， Navigation Drawer 和 Mega Menu 都在主界面上覆盖了一个 layer ，而这个 layer 的大小足以让设计师把这里好好弄弄了，这两种导航模式相比， Mega Menu 的发挥空间甚至更大，所以重拾这个古早的导航模式在应用层级复杂、导航功能拓充需求强烈的现在，未必不是件好事。



原文链接:http://www.catyeah.com/blog/?p=724

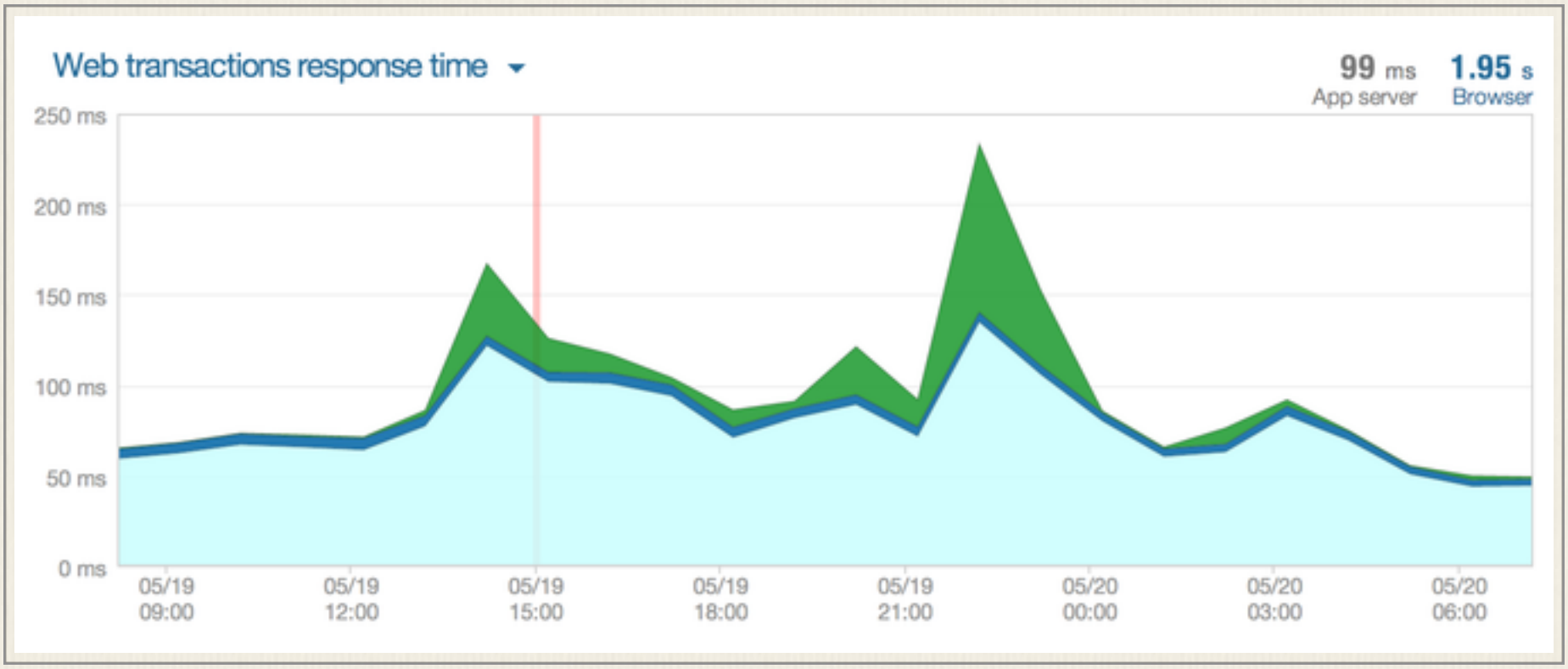
Ruby China 里面我是如何设计缓存的

作者:李华顺

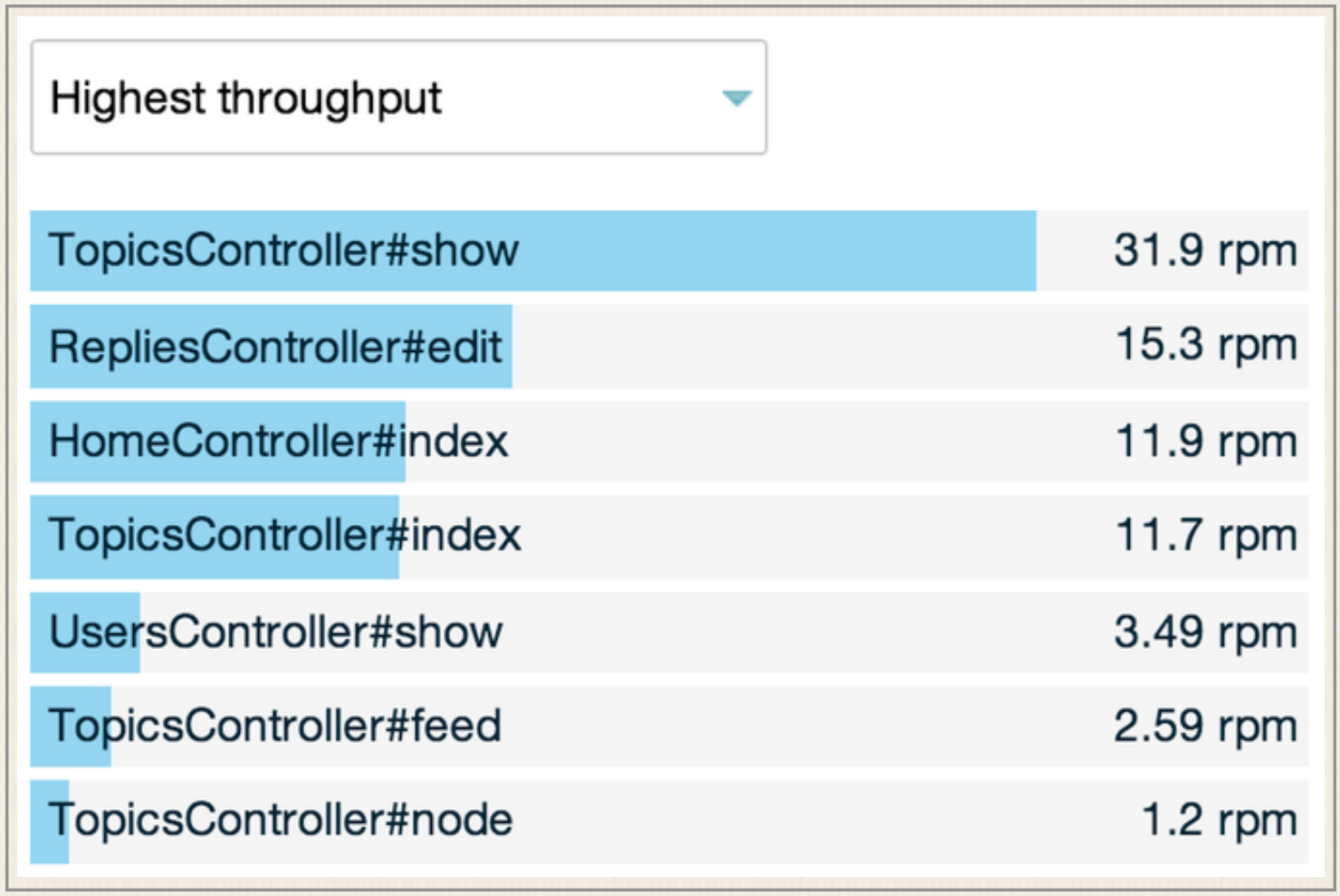
看最近 @quakewang 分享的 《总结 web 应用中常用的各种 cache》，我也搭车分享一下在 Ruby China 里面，我们是如何做 Cache 的。

首先给大家看一下 NewRelic 的报表

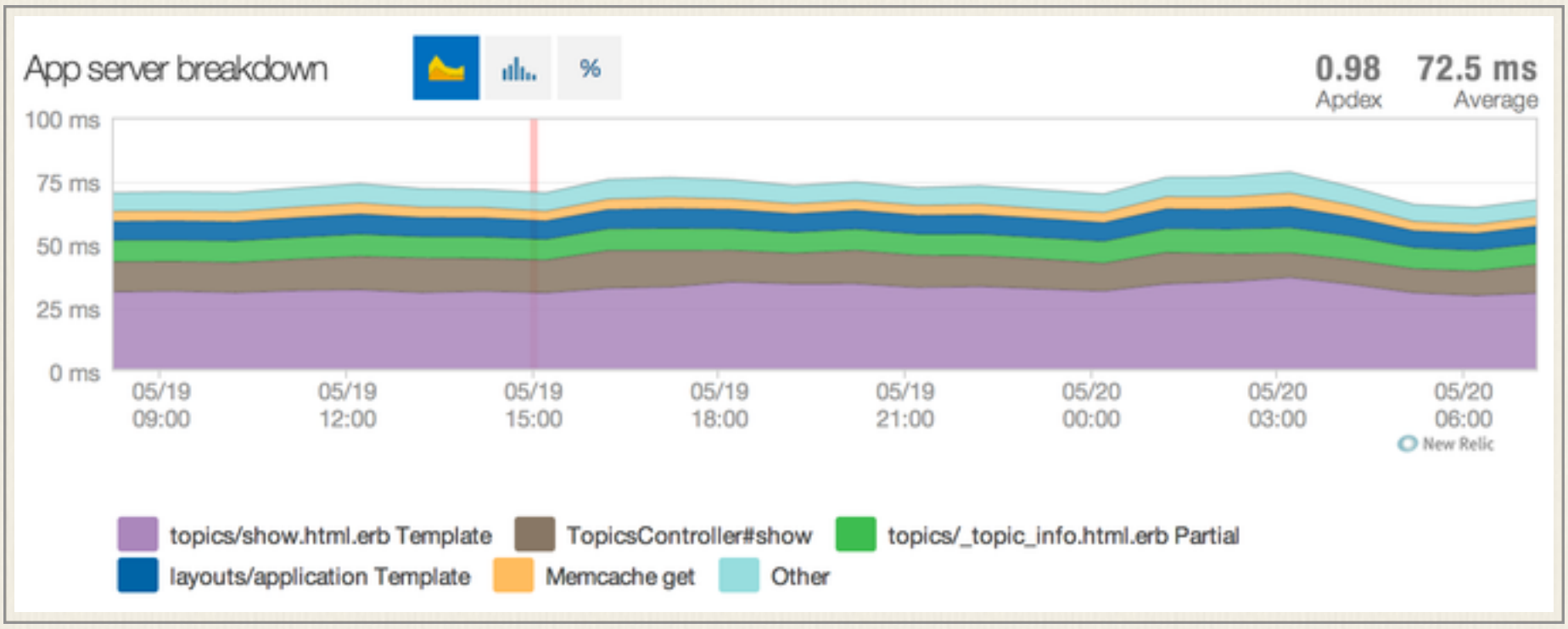
最近 24h 的平均响应时间



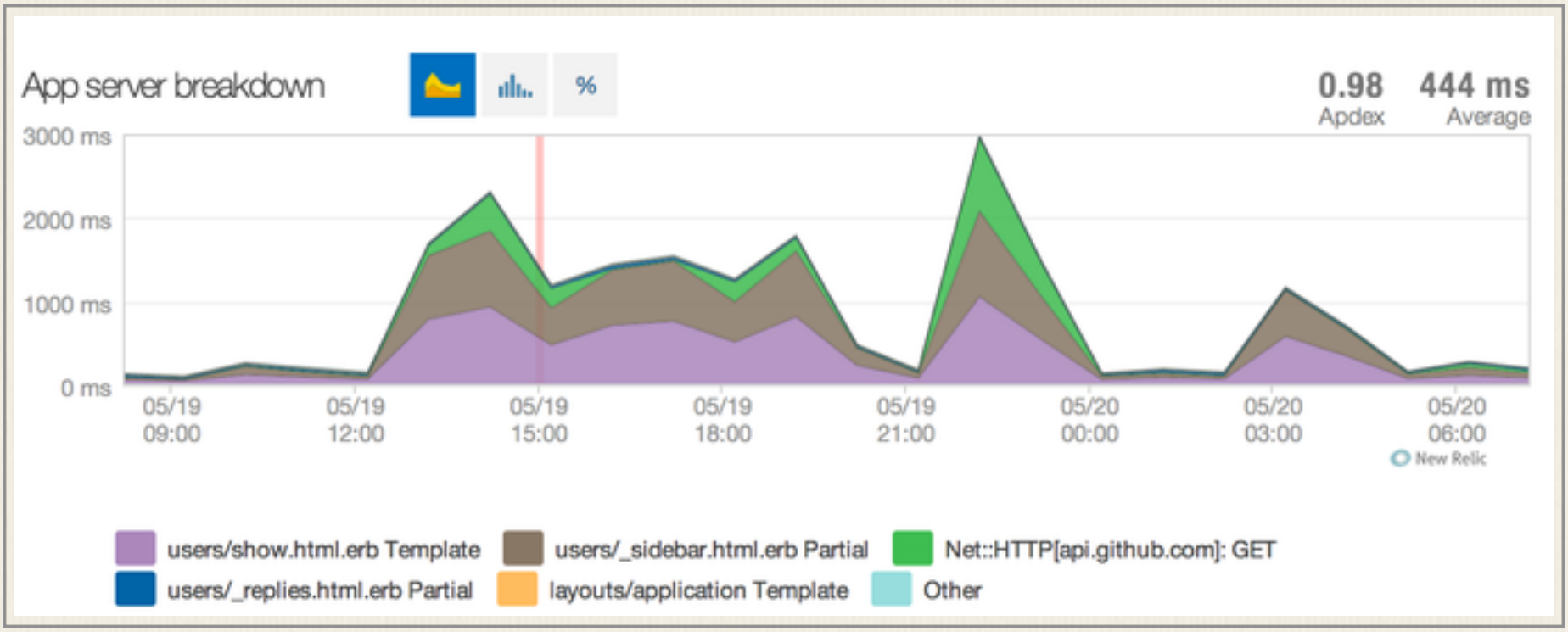
流量高的那些页面 (Action)



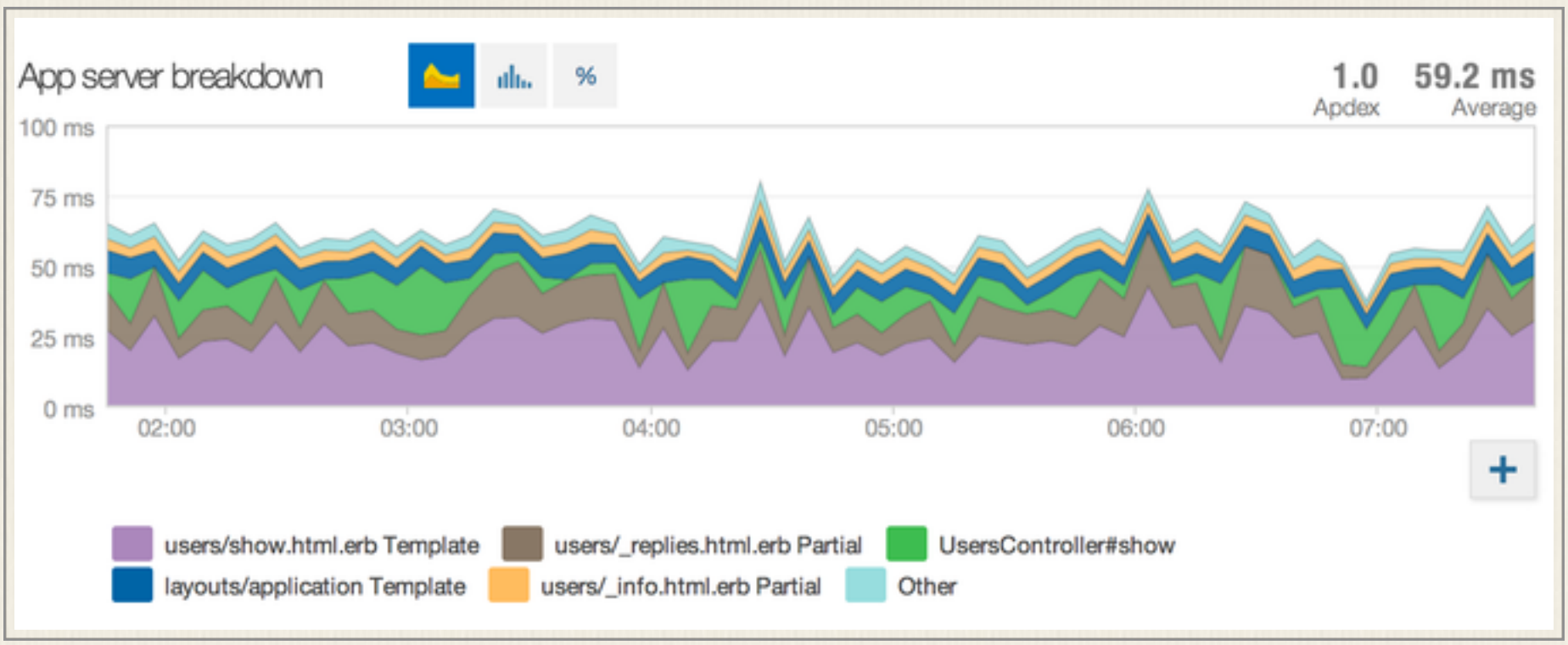
访问量高的几个 **Action** 的情况：



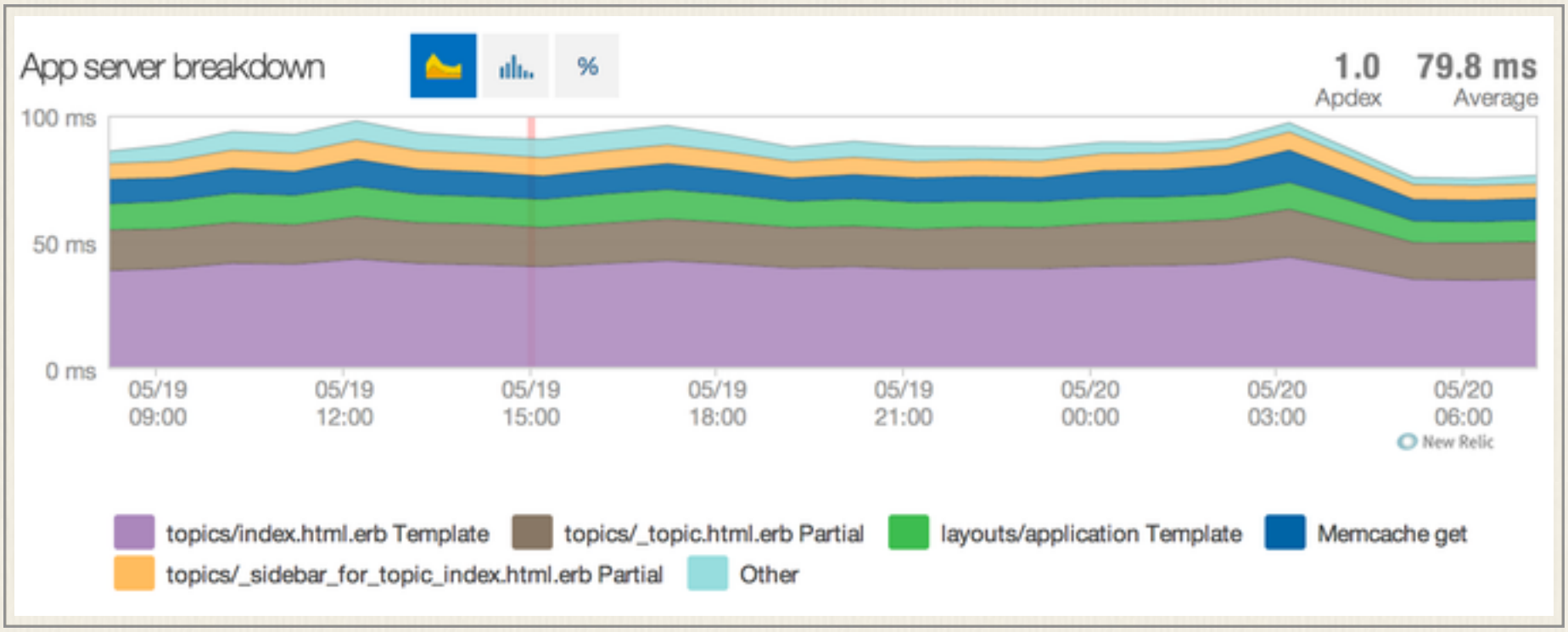
UsersController#show (比较惨，主要是 GitHub API 请求拖慢)



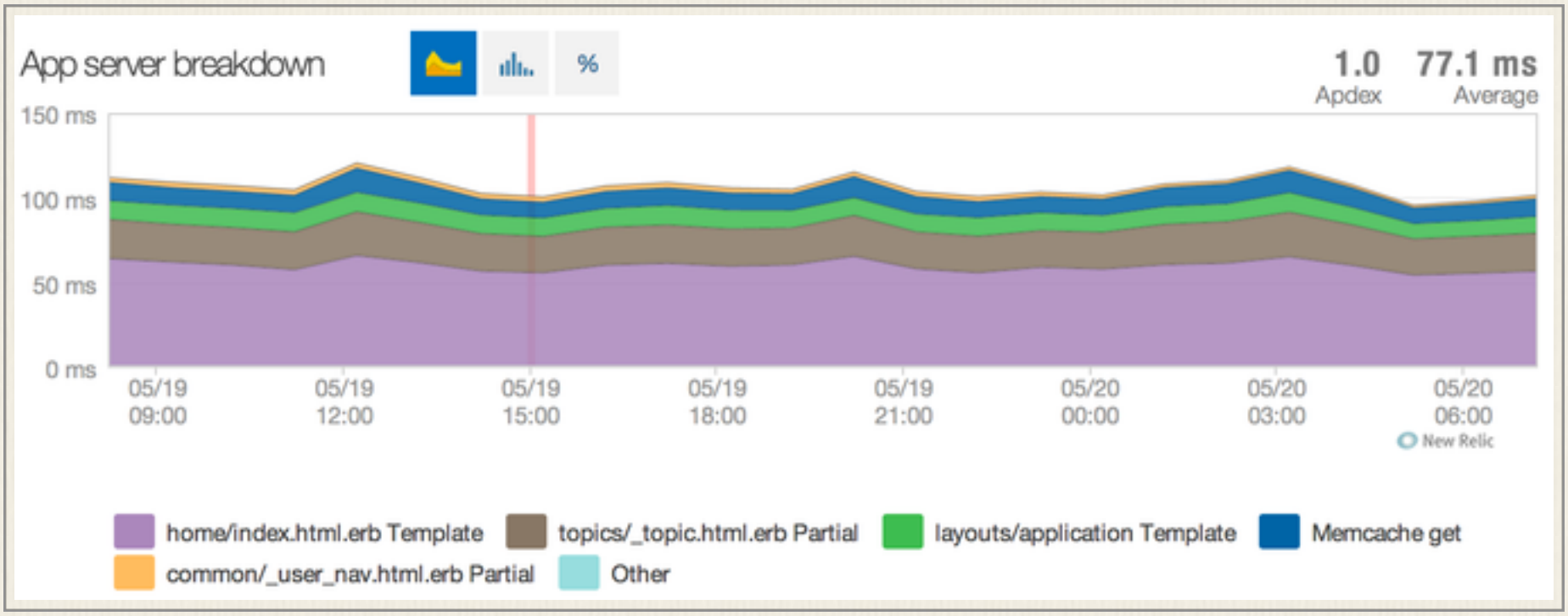
PS: 在发布这篇文章之前我有稍加修改了一下，**GitHub** 请求放到后台队列处理，新的结果是这样:



TopicsController#index



HomeController#index



从上面的报表来看，目前 Ruby China 后端的请求，排除用户主页之外，响应时间都在 100ms 以内，甚至更低。

我们是如何做到的?

- Markdown 缓存
- Fragment Cache
- 数据缓存
- ETag
- 静态资源缓存 (JS,CSS,图片)

Markdown 缓存

在内容修改的时候就算好 Markdown 的结果，存到数据库，避免浏览的时候反复计算。

此外这个东西也特意不放到 Cache，而是放到数据库里面：

1. 为了持久化，避免 Memcached 停掉的时候，大量丢失；
2. 避免过多占用缓存内存；

```ruby
class Topic
  field :body # 存放原始内容，用于修改
  field :body_html # 存放计算好的结果，用于显示

  before_save :markdown_body
  def markdown_body
    self.body_html = MarkdownTopicConverter.format(self.body) if self.body_changed?
  end
end
```

Fragment Cache

这个是 Ruby China 里面用得最多的缓存方案，也是速度提升的原因所在。

app/views/topics/_topic.html.erb

```erb
<% cache([topic, suggest]) do %>
<div class="topic topic_line topic_<%= topic.id %>">
   <%= link_to(topic.replies_count,"#{topic_path(topic)}#reply#{topic.replies_count}",
         :class => "count state_false") %>
  ... 省略内容部分

</div>
<% end %>
```

1. 用 topic 的 cache_key 作为缓存 cache views/topics/{编号}-#{更新时间}/{suggest 参数}/{文件内容 MD5} -> *views/topics/19105-20140508153844/false/bc178d556ecaee49971b0e80b3566f12*

2. 某些涉及到根据用户帐号，有不同状态显示的地方，直接把完整 HTML 准备好，通过 JS 控制状态，比如目前的“喜欢“功能。

```erb
<script type="text/javascript">
  var readed_topic_ids = <%= current_user.filter_readed_topics(@topics) %>;
  for (var i = 0; i < readed_topic_ids.length; i++) {
    topic_id = readed_topic_ids[i];
    $(".topic_"+ topic_id + " .right_info .count").addClass("state_true");
  }
</script>
```

再比如app/views/topics/_reply.html.erb

```erb
<% cache([reply,"raw:#{@show_raw}"]) do %>
<div class="reply">
  <div class="pull-left face"><%= user_avatar_tag(reply.user, :normal) %></div>
  <div class="infos">
    <div class="info">
      <span class="name">
        <%= user_name_tag(reply.user) %>
      </span>
      <span class="opts">
        <%= likeable_tag(reply, :cache => true) %>
        <%= link_to("", edit_topic_reply_path(@topic,reply), :class => "edit icon small_edit", 'data-uid' => reply.user_id, :title => "修改回帖")%>
        <%= link_to("", "#", 'data-floor' => floor, 'data-login' => reply.user_login,
            :title => t("topics.reply_this_floor"), :class => "icon small_reply" )
        %>
      </span>
    </div>
    <div class="body">
      <%= sanitize_reply reply.body_html %>
    </div>
  </div>
</div>
<% end %>
```

同样也是通过 reply 的 cache_key 来缓存 views/replies/202695-20140508081517/raw:false/d91dddbcb269f3e0172bf5d0d27e9088 同时这里还有复杂的用户权限控制，用 JS 实现；

```erb
<script type="text/javascript">
  $(document).ready(function(){
    <% if admin? %>
      $("#replies .reply a.edit").css('display','inline-block');
    <% elsif current_user %>
      $("#replies .reply a.edit[data-uid='<%= current_user.id %>']").css('display','inline-block');
    <% end %>
    <% if current_user && !@user_liked_reply_ids.blank? %>
      Topics.checkRepliesLikeStatus([<%= @user_liked_reply_ids.join(",") %>]);
    <% end %>
  })
</script>
```

数据缓存

其实 Ruby China 的大多数 Model 查询都没有上 Cache 的，因为据实际状况来看，MongoDB 的查询响应时间都是很快的，大部分场景都是在 5ms 以内，甚至更低。

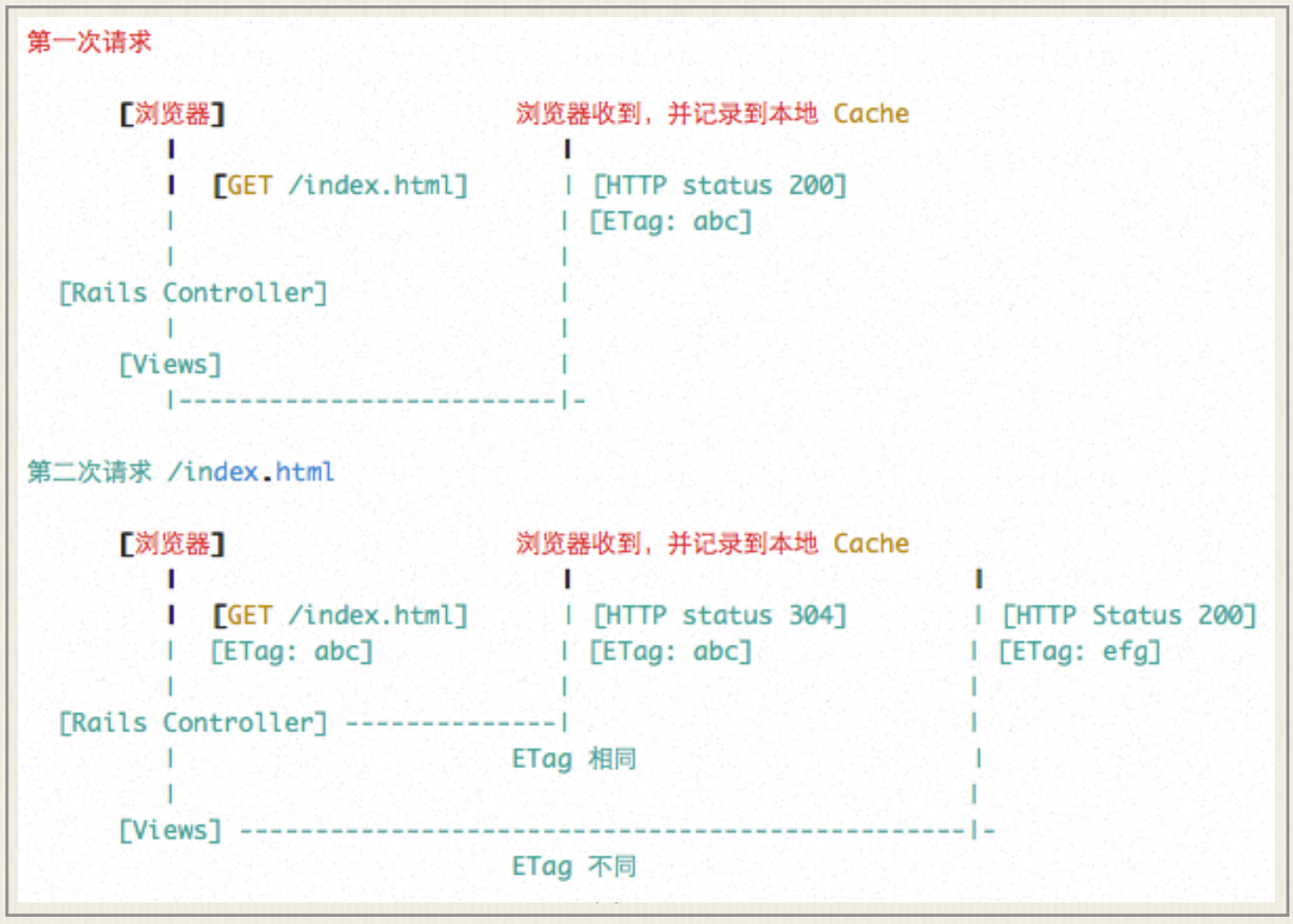
我们会做一些比价负责的数据查询缓存，比如：GitHub Repos 获取

```ruby
def github_repos(user_id)
  cache_key = "user:#{user_id}:github_repos"
  items = Rails.cache.read(cache_key)
  if items.blank?
    items = real_fetch_from_github()
    Rails.cache.write(cache_key, items, expires_in: 15.days)
  end
  return items
end
```

ETag

ETag 是在 HTTP Request, Response 可以带上的一个参数，用于检测内容是否有更新过，以减少网络开销。

过程大概是这样



Rails 的 fresh_when 方法可以帮助将你的查询内容生成 ETag 信息

```
def show
  @topic = Topic.find(params[:id])

  fresh_when(etag: [@topic])
end
```

静态资源缓存

请不要小看这个东西，后端写得再快，也有可能被这些拖慢（浏览器上面的表现）*!*

1、合理利用 *Rails Assets Pipeline*，一定要开启*!*

```
# config/environments/production.rb
config.assets.digest = true
```

2、在 *Nginx* 里面将 *CSS, JS, Image* 的缓存有效期设成 *max*；

```
location ~ (/assets|/favicon.ico|/*.txt) {
  access_log        off;
  expires           max;
  gzip_static on;
}
```

3、尽可能的减少一个页面 *JS, CSS, Image* 的数量，简单的方法是合并它们，减少 *HTTP* 请求开销；

<head>

...

只有两个

<link href="http://l.ruby-china.org/assets/front-1a909fc4f255c12c1b613b3fe373e527.css" rel="stylesheet" />

<script src="http://l.ruby-china.org/assets/app-24d4280cc6fda926e73419c126c71206.js"></script>

...

</head>

一些 *Tips*

1. 看统计日志，优先处理流量高的页面；

2. *updated_at* 是一个非常有利于帮助你清理缓存的东西，善用它！修改数据的时候别忽略它！

3. 多关注你的 *Rails Log* 里面的查询时间，*100ms* 一下的页面响应时间是一个比较好的状态，超过 *200ms* 用户就会感觉到迟钝了。

原文链接:http://huacnlee.com/blog/cache-design-in-ruby-china/

Windows平台分布式架构实践 - 负载均衡

作者: Jesse Liu

概述

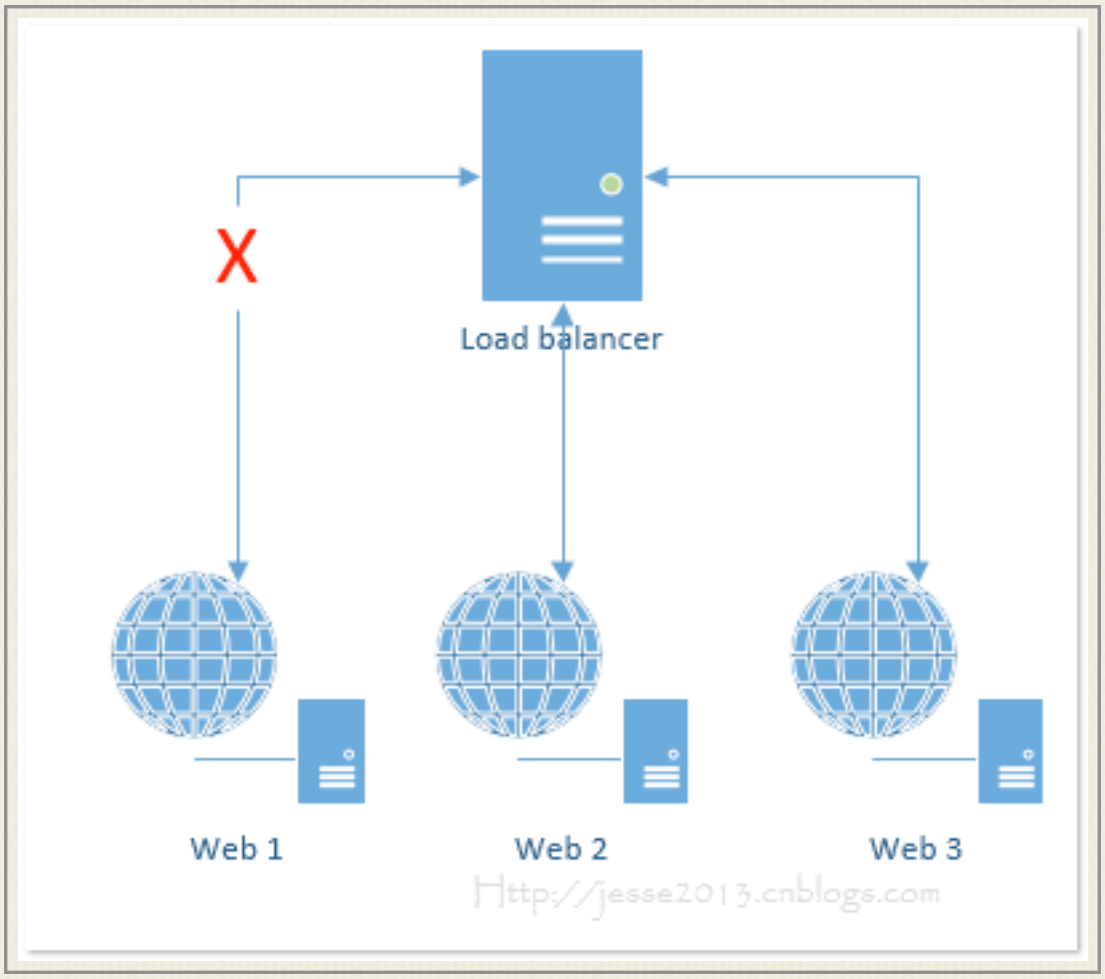
最近.NET的世界开始闹腾了，微软官方终于加入到了对.NET跨平台的支持，并且在不久的将来，我们在VS里面写的代码可能就可以通过Mono直接在Linux和Mac上运行。那么大家（开发者和企业）为什么那么的迫切的希望.NET跨平台呢？第一个理由是便宜，淘宝号称4万多台服务器全部运行在Linux，Linux平台下还有免费的MySql,这些都是免费的，这些省下来直接就是利润呀，做企业的成本可以降低又没有任何损失，何乐而不为呢？第二个理由是在Linux系统下还有很多非常优秀的构架（当然同样也是免费的），分布式缓存Memcached, 大数据处理构架Hadoop等等，这些都为一些大型的分布式系统提供了很好的支撑，当然还有诸如Liniux系统本身的一些安全和网络方面的优势，等等。 所以也难怪大佬们都纷纷不约而同的没有选择.NET。

但是如果.NET也支持跨平台之后，那这样的格局可能就要发生变化了。上面所有的优势依然可以保留，并且加上它语法的优越性，以及快速的开发效率等，还是会为其争得一席之地的。

但是，是不是Windows平台下就不能实现这些大型的分布式系统呢？我相信这个问题已经被广泛讨论过，但是至少我没有看到比较清晰的，完整的案例。带着这些问题，我决定升级我的机器，自己从头到尾在windows平台下搭建一个高可扩展性的分布式网站出来。我经验尚浅，很多的东西还处于摸索阶段，所以如果有错误，还请大师多多指点。

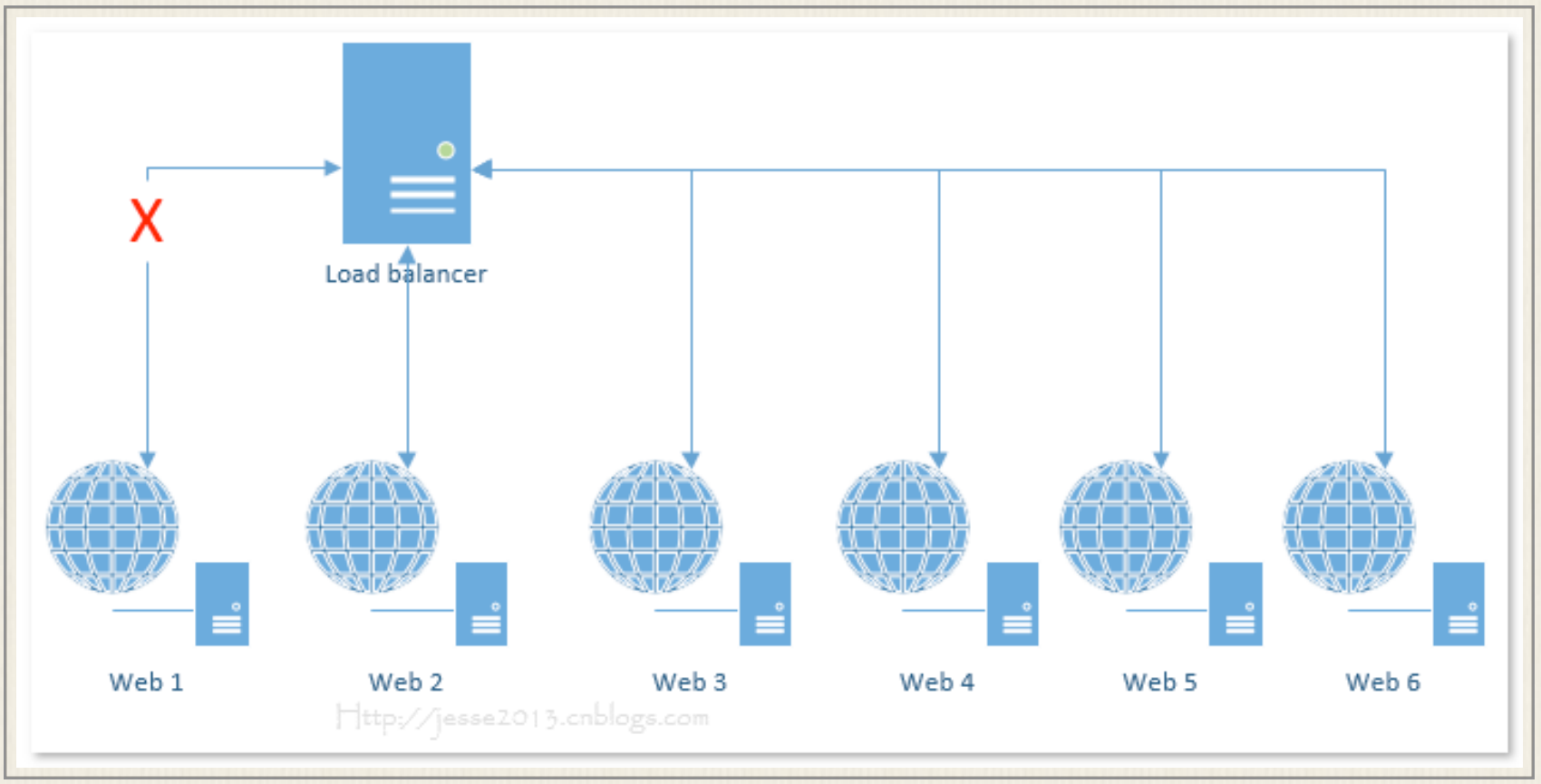
什么是负载均衡

负载均衡可以帮我们解决两个方面的问题，第一个即提高可用性。这里面的可用性主要是从WEB服务器，的角度来讲的，如果说我们只有一台Web服务器，而它遇到了某种未知的错误导致IIS无法启动，那么我们的网站就无法访问了，这就是一种比较低的可用性。那么利用负载均衡，放在我们Web服务器的前面，由它来收集所有的请求，然后转发给我们的Web服务器， 这时候我们就可以添加两台Web服务器，如果其中有一台坏了，至少还有另一台在工作，也不至于导致我们的网络无法访问。



当然，有人可能会问，如果那台Load balancer坏了怎么办？那不是还是访问不了网站么？我们这里讨论的是提高可用性，在做到365天*24小时不间断的服务，需要有另外的组件来支撑，我们留在后面讨论。除了可用性以外，负载均衡还可以帮助我们提高可扩展性，当然这个可扩展性同样是指的Web服务器层面。从网站性能的角度来讲，好几个程序员花上好几天的

时间做了一些优化所带来的效果有时候可能还没有直接加一根内存条来的快。内存加完了没什么影响，我们还可以换更好的CPU，CPU换完了，我们还可以用固态硬盘，甚至很多公司已经开始直接把数据放到内存中了（注：具体场景具体对待）。 如果这些都不可以再加了呢？那就再加机器吧，一台服务器可以处理1000个并发，那么两台理论上是2000了，所以这就有了我们的横向扩展。



负责均衡器分发请求的类型

所有的请求首先全部到达Load balancer ，再由它转发到具体的Web服务器，转发的方式分为以下几种：

- 轮转调度(Round-robin):最简单的方式，这种方式基本上不能算是负载均衡。第一个请求给web1，下一个给web2，再下一个给web3... 不会考虑某 一个服务器是不是负荷太重等等。
- 基于权重的分配(Weight-based): 可以配置每一台服务器处理请求数据的比例，特别适合那种有某台服务器配置不一样的场景。比如说某台服务器配置特别好，那我们可以让它多处理一些请求。
- 随机（Random): 随机分配。

- 粘性session（Sticky Session): Load balancer会跟踪请求，确保同一个session id的请求都交给同一样服务器。
- 最空闲优先（Least current request ）：将最新的请求转发给当前处理请求数量最小的那个服务器。
- 响应时间优先(Response time)：哪台服务器当前响应时间最短就给哪台。
- 用户或URL参数选择(User or URL information)：部分负载均衡器还提供根据一些参数来决定哪台服务器来处理，比如说根据用户信息，地址位置，URL参数，cookie信息等 。

我们还可以根据负载均衡器所使用的技术将它们分为以下几类：

- 反向代理：负载均衡器作为一个代理，同时维持着两个TCP请求，从客户端接收请求，然后将请求转发给相应的Web 服务器,等Web返回Response的时候是返回给了代理服务器，然后再由代理服务器转交给真正的客户端。这样就会导致有一些功能不可用，比如在WEB服务器环境查看请求的来源IP实际上成了我们代理服务器的IP等。
- 透明反向代理：和上面的代理服务器一样，只不过WEB服务器从Request中获取到的信息是真正客户端的信息，就是好像没有使用代理一样的。
- 直接服务器返回：通过更改WEB服务器的MAC 地址来实现分发请求，在这种方式下，WEB服务器不会像上面使用代理服务器一样，请求处理完之后是直接返回给客户端的，所有相对于反向代理来说这种方式的性能会更快一些。
- NAT 负载均衡：NAT(Network Address Translation网络地址转换)，将网络包（可以理解成TCP包）中的目标IP地址变成实现要处理这个请求的WEB服务器的地址。
- Microsoft 网络负载均衡：Windows 自带的负载均衡组件，一会我们就用它来做测试。

不使用负载均衡的测试结果

一台独立的服务器

我们可以从一个网站的最初级版本开始说起，最开始的时候我们决定搭建一个网站，但是我们也不知道效果会怎么样，光键是那时候，我们很穷，于是我们租用了一台托管主机，它可能承担了至少三个或以上的角色：WEB服务器、静态资源服务器，以及数据库服务器。我们可以用ASP.NET MVC4 + SQL 2008来做一个基本的电子商务网站，基本够用了。但是能够承载多大的访问量呢？下面我们来做一个简单的测试（注意：本文以后本系列所面所有的测试都是在虚拟机上进行的，忽略网络的因素，以及多台虚拟机同时运行时CPU资源的因素，所以测试结果只是一个参考）。

在我的机器上有一台虚拟机配置如下：

CPU: Intel Core I5- 4570, 3.19GHz,
内存: 4G
硬盘： 20G (ShineDisk 固态硬盘)

测试页面没有什么复杂的逻辑，利用ASP.NET MVC4 + Entityframe-work 6.0 + SQL 2008 + IIS8.5来实现， 我们的页面也只是一个简单的列表页，列出系统里面所有的商品。

Home Controller 代码

```
public class HomeController : Controller
{
    private CarolContext db = new CarolContext();

    0 references
    public ActionResult Index()
    {
        return View(db.Products.ToList());
    }
}
```

Index.cshtml 代码

```
@model IEnumerable<Carol.Web.Models.Product>
@{
    ViewBag.Title = "Home Page";
}
<div class="content">
    @foreach (var p in Model)
    {
        <div class="list-item">
            <ul>
                <li>
                    <img src="@Url.ListingThumb(p.Id,p.ListingThumbName)" />
                </li>
                <li><a>@p.Title</a></li>
                <li>@p.Price</li>
            </ul>
        </div>
    }
</div>
```

在数据库初始化的时候插入*500*条测试数据

```
public class CarolInitializer : DropCreateDatabaseIfModelChanges<CarolContext>
{
    0 references
    protected override void Seed(CarolContext context)
    {
        var random = new Random(500);
        for (var i = 0; i < 300; i++)
        {
            var product = new Product
            {
                Title = string.Format("Product {0}", i.ToString()),
                Price = random.Next(1000) * i,
            };

            context.Products.Add(product);
            context.SaveChanges();
        }
    }
}
```

连接字符串就使用本地连接就可以了。

<connectionStrings>

<add name="CarolContext" connectionString="Server=localhost;database=carol;trusted_connection=true" providerName="System.Data.SqlClient" />

</connectionStrings>

我们使用的轻量级的*ab*来做压力测试，如果不熟悉*ab*的可以点这里，下面是测试的结果：

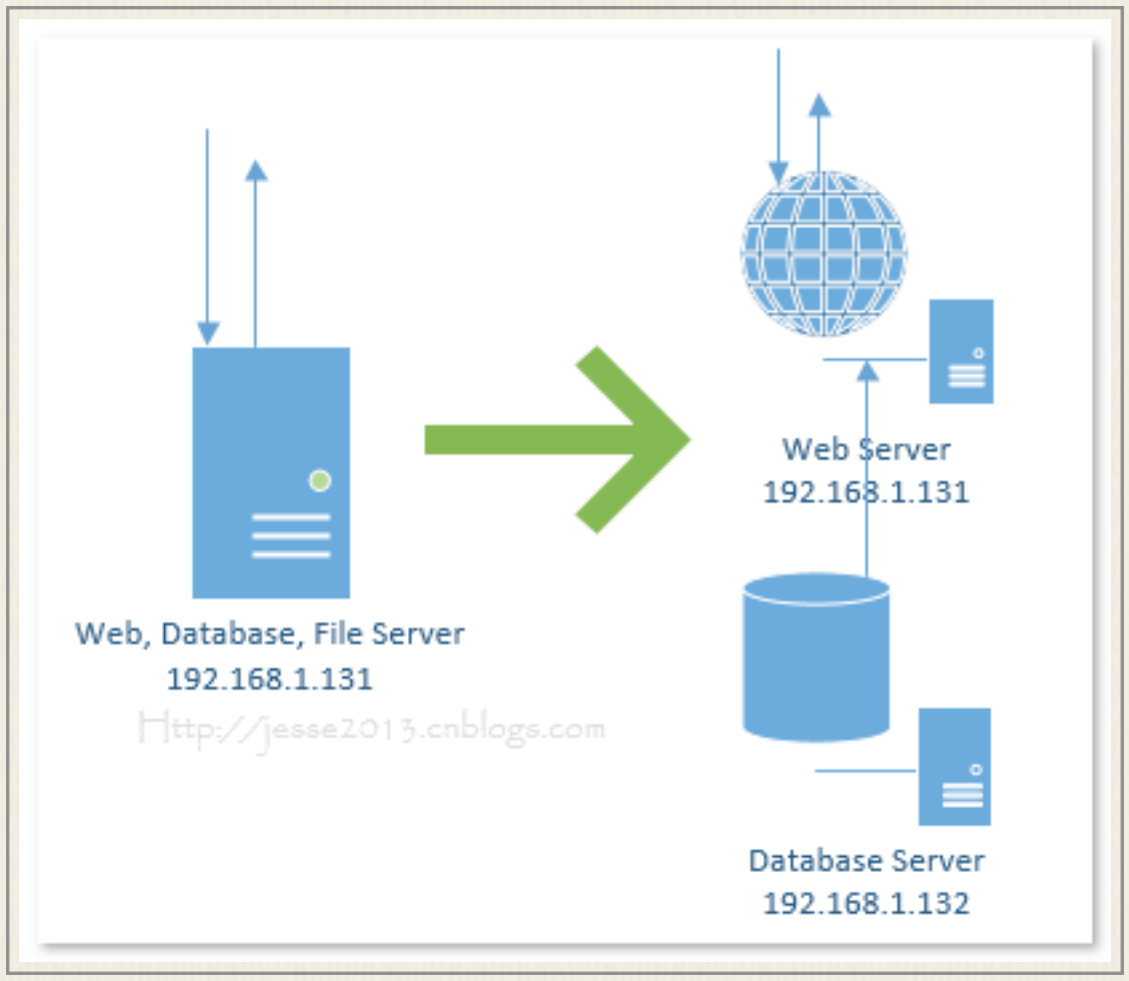
ab -n1000 -c100 http://192.168.1.131

总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	113.98	87.732
1000	100	115.71	864.216
2000	100	119.90	833.96
1000	200	122.96	1583.53
2000	200	118.38	1576.53
1000	300	128.91	2327.259
1000	400	127.53	3136.637

通过测试发现，我们这单个服务器的吞吐率接近在*110~130*之间，而一旦并发数达到*200*以后，每个请求的处理时间就达到*1.5s*多了，*400*个并发用户的时候每个请求要花上*3s*多的时间。如果在真实的网络环境中可能会更差。由此我们可以得出我们这个服务器可能最大支持*120*人左右同时访问。

*WEB*服务器与数据库服务器分离

现在我们来做一个花费不是很大，又空间做的扩展，也不需要改任何架构，我们只是再加一台专门的数据库服务器。



下面我们再来看一下测试结果：

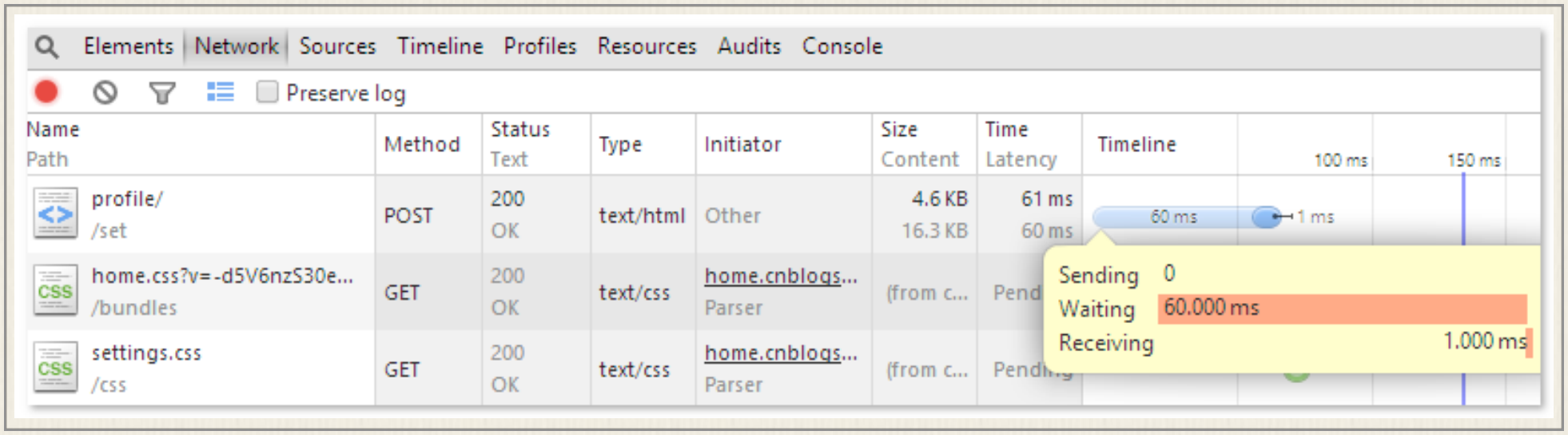
总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	145.62	68.670
1000	100	150.17	665.934
2000	100	151.94	658.146
1000	200	156.59	1277.205
2000	200	160.48	1246.286
1000	300	150.61	1991.916
1000	400	154.88	2582.637

大家可以看到，这里我们的吞吐率(每秒处理请求数已经提升到了150左右)，并发处理能力提升了50%，并且300和400并发的时候响应时间也比上面的架构要好一些。

使用负载均衡的测试结果

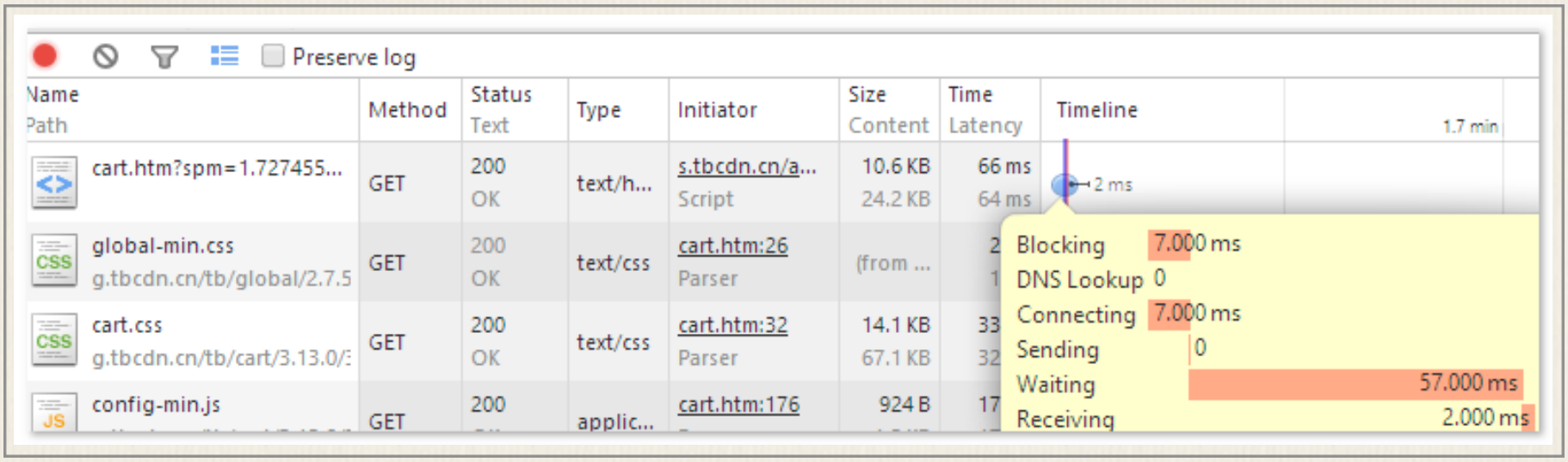
安装网络负载均衡（NLB)

上面我们一台独立的Web服务器和一台独立的数据库服务器的组合已经可以处理150左右的并发了，现在我们假想一下如果网站的的知名度越来越大，如果同时有400个用户来访问怎么办？ 从上面的图中我们可以看到400个并发的时候服务器的处理时间为2582.637ms（实现上这是拿到响应的时间，因为我们是一台机器上的不同虚拟机，我是在主机上做测试，所以我们就忽略网络传输的时间，假设这个就是我们的服务器处理时间），这个服务器响应时间也就是我们通过F12->网络 中看到的等待时间 。



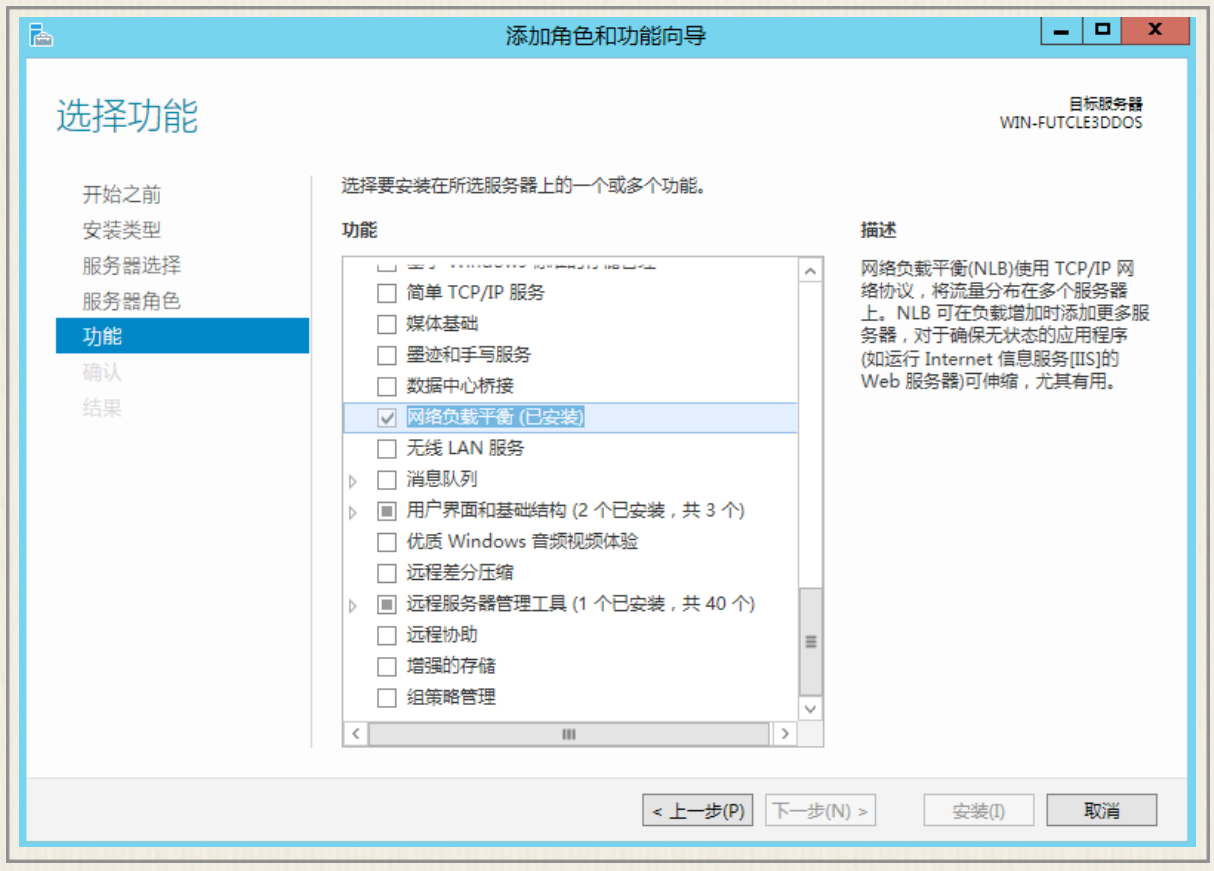
页面什么时候能拿到这个响应还要加上网络传输的时间，也就是Receiving。1ms的传输时间堪称神速啊！我家用的长城宽带10M，总是早上网络出奇的好，一到晚上就挂掉了，还有可能就是你们现在都没有上博客园 :)

用户体验黄金法则之一: 网站加载时间 = 用户体验，别说3S，可能等个2S你页面还不出来，用户准备离开了，下面是淘宝购物车页面的加载时间 。



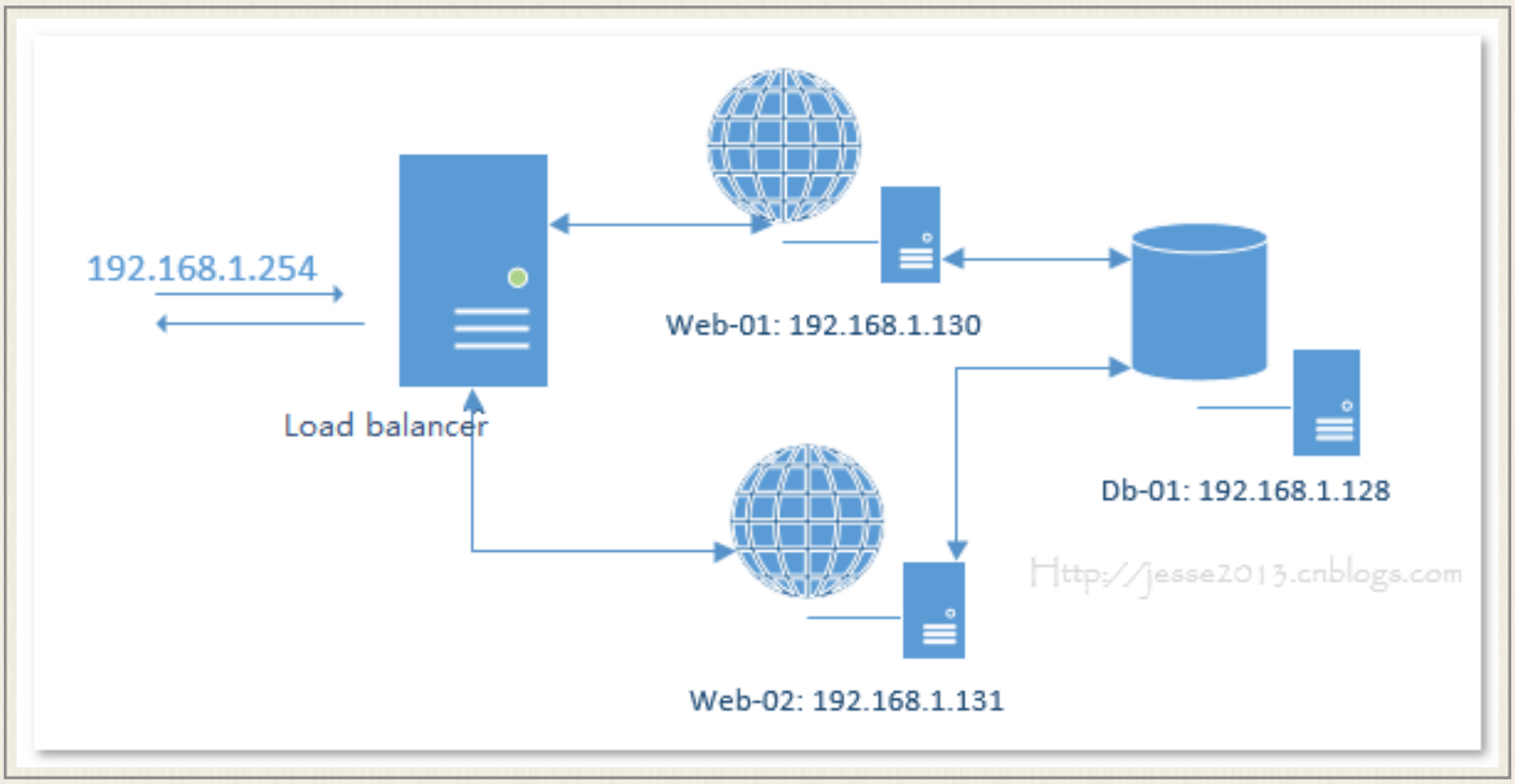
国内很多大型的网站的响应时间基本上都控制在100ms以内，基本达到那种一输入地址敲回车，眨眼之间页面就出来了。当然这并不是光有个负载均衡加几台web服务器就能解决的，我们后来再来一步一步的实践下去。话说回来，我们上面的测试结果基本上只有并发为10的时候响应时间是在100ms以内的， 看来我们还有很长的一段路要走啊。

正所谓“最好的架构是进化而来的，而不是设计出来的”，面对我们现在的瓶颈暂时通过负载均衡添加多台Web服务器就可以了。我们上面讲到负载均衡器类型的时候有一种 Microsoft负载均衡，我们可以很轻松的通过服务器管理器来将这些组件安装到我们的服务器中。 安装我们就不讲了，就是通过服务器管理-> 添加角色和功能->在功能中选择“网络负载均衡” 然后安装就可以了。



注意：图中的Load balancer实际上是不存在的，因为只要我们2台Web服务器安装了网络负载平衡组件，在其中任意一台上建立群集就可以了，图是为了方便大家理解。

这样的话我们的服务器架构就成了下面这个样子：

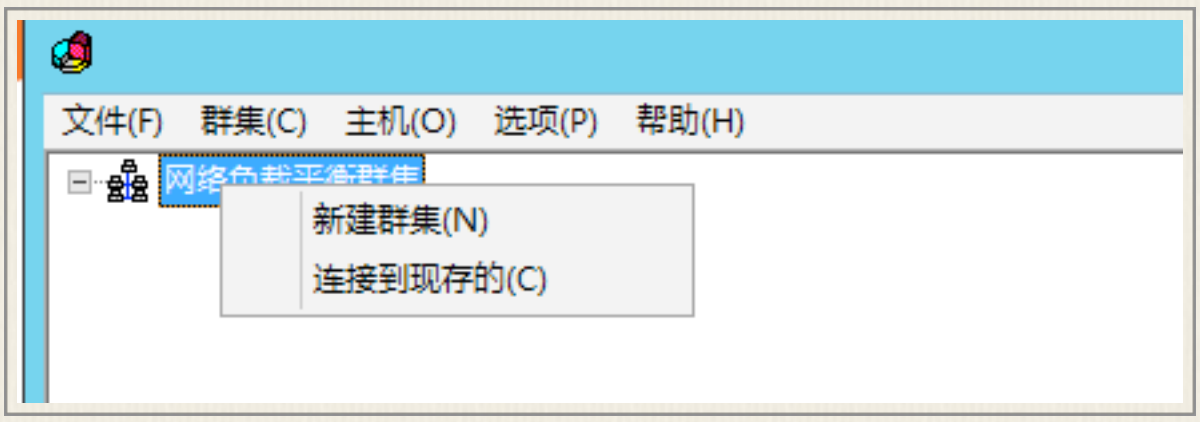


192.168.1.254 就是我们暴露的外部IP地址，访问192.168.1.254的请求就会转发给后面的两台WEB服务器。

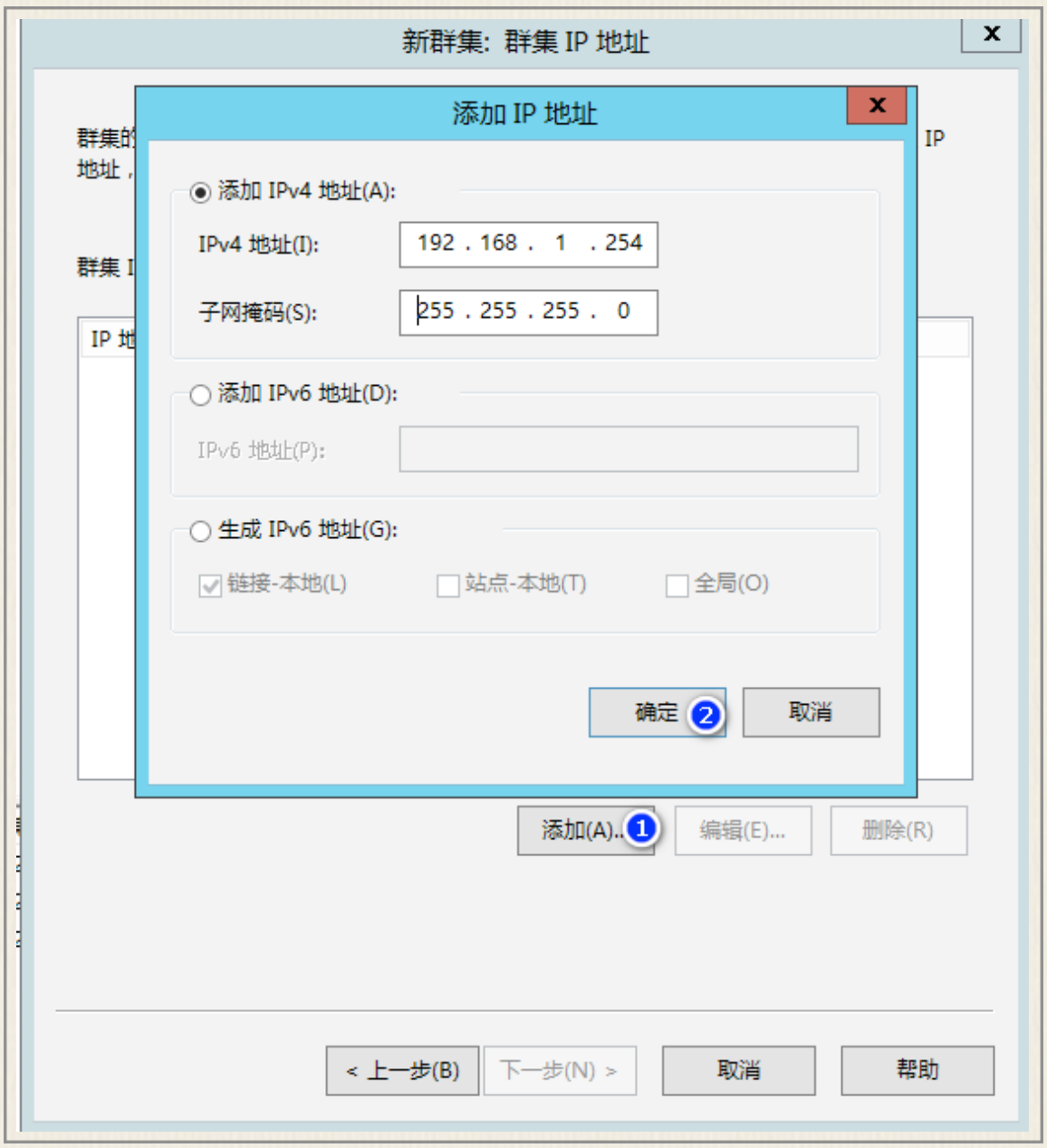
配置网络负载均衡

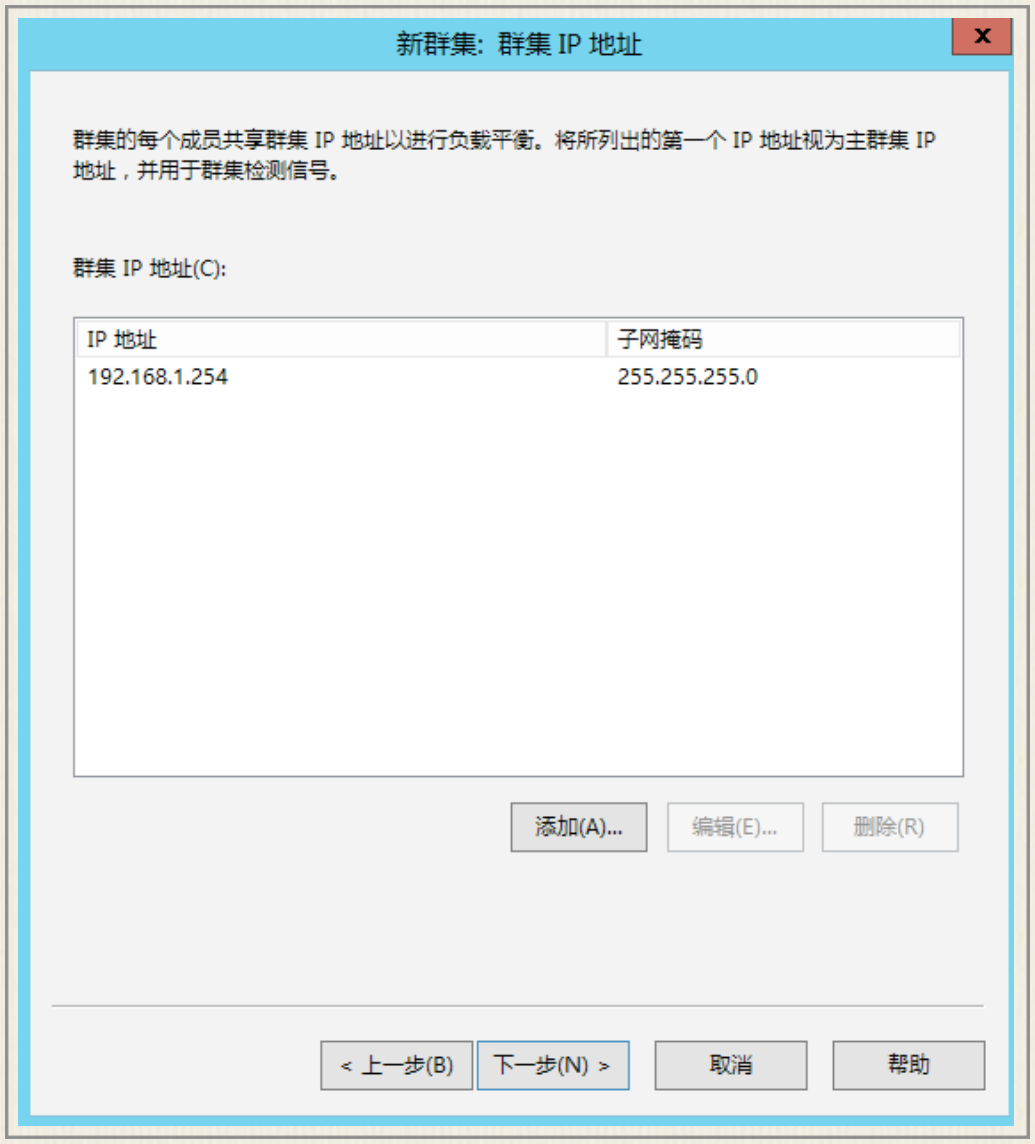
在我们为上面2台WEB服务器安装NLB之后，我们在其中任意一台上来新建群集，然后将另外一台加入到这个群集中即可，我们就在web-01(192.168.1.130)上来新建这个群集。在建立群集之前，我们要确保这2台服务器都是使用的静态IP，否则无法将他们加入到群集中。

- 在web-01(192.168.1.130)上从管理工具中打开 网络负载均衡器
- 右击“网络负载平衡群集”，选择“新建群集”



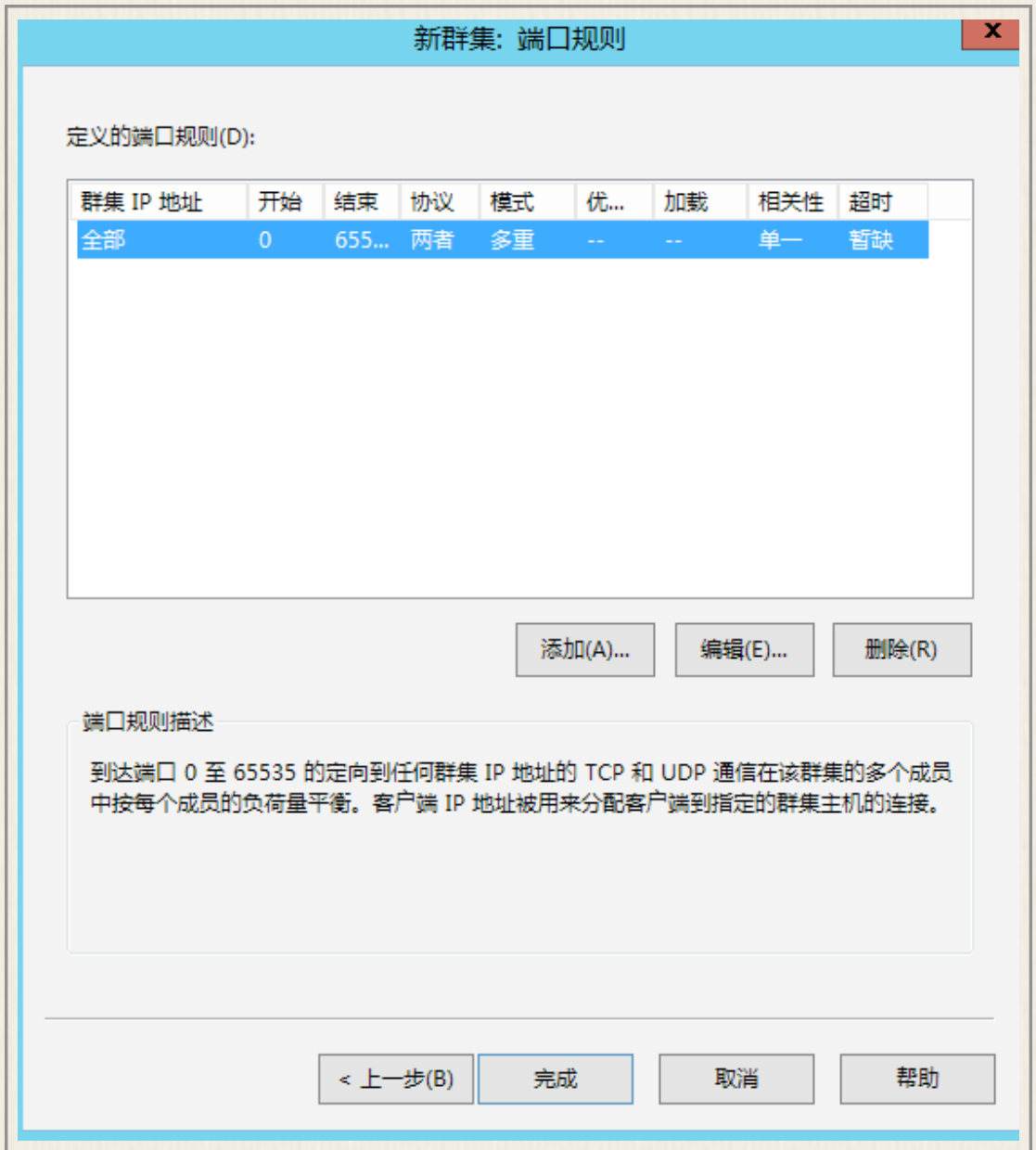
- 在“新群集：连接”窗口中将 192.168.1.130添加为主机，点击下一步
- 进入 “新群集：主机参数”，直接下一步
- 进入 “新群集：群集IP地址”， 添加窗口中的“添加” 将192.168.1.254添加到窗口中然后点击下一步





- 进入 “新群集：群集参数”，选择“多播”然后点击下一步

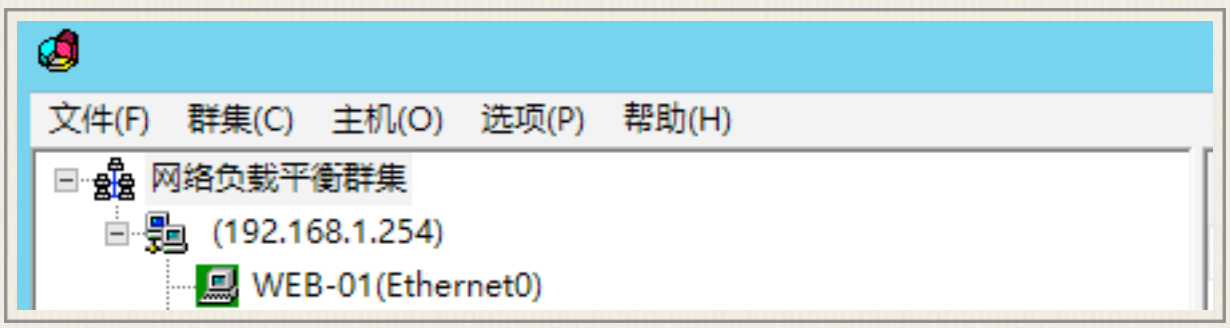
- 进入 “新群集：端口规则”，选中全部，然后点击编辑



将端口范围改成 80~80，协议选 “TCP”，相关性选“无”

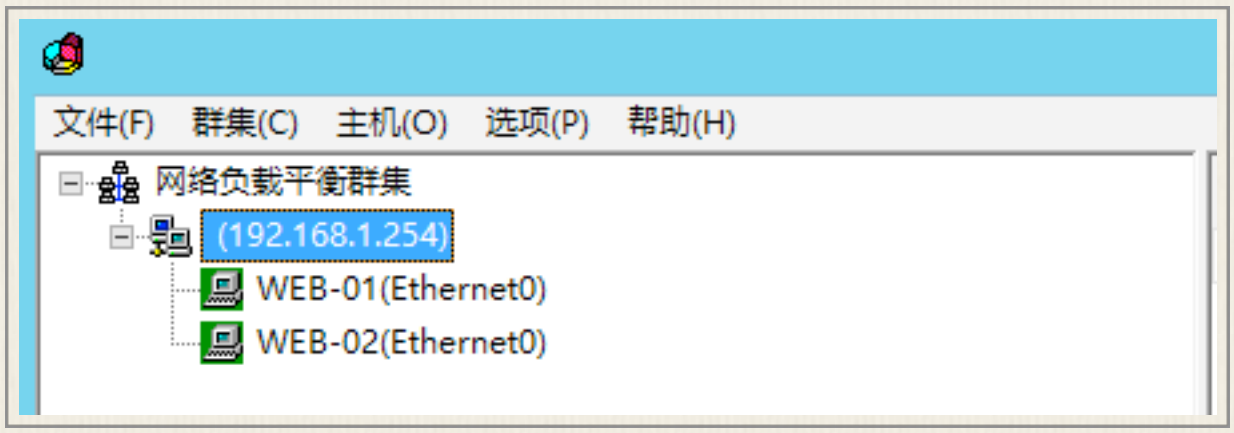
- 点击确定回到主窗口，然后点击完成。

• 通过上面的步骤，我们已经建立了一个群集，并且将web-01 加入到了群集中，我们还需要手动将web-02也加入到群集中。



• 在群集(192.168.1.254)上右键点击“添加主机到群集”

• 在“将主机添加到群集：连接”窗口中的 主机中输入192.168.1.131然后后面一下点下一步即可。



现在我们就可以到我们的真实机器上去访问192.168.1.254了，也就是说马上我们就进入测试环节了。

测试结果

本文中所有的测试结果都没有取第一次的结果，EF也需要预热，同样的查询在EF中也是有缓存的，所以第一次的结果会与后面的测试结果有很大的区别，后面的几次（在相同参数下）基本差别就不大了。

总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	227.76	43.907
1000	100	205.75	486.026
2000	100	230.22	434.365
1000	200	221.89	901.336
2000	200	263.46	759.141
1000	300	206.87	1450.194
1000	400	207.6	1931.764

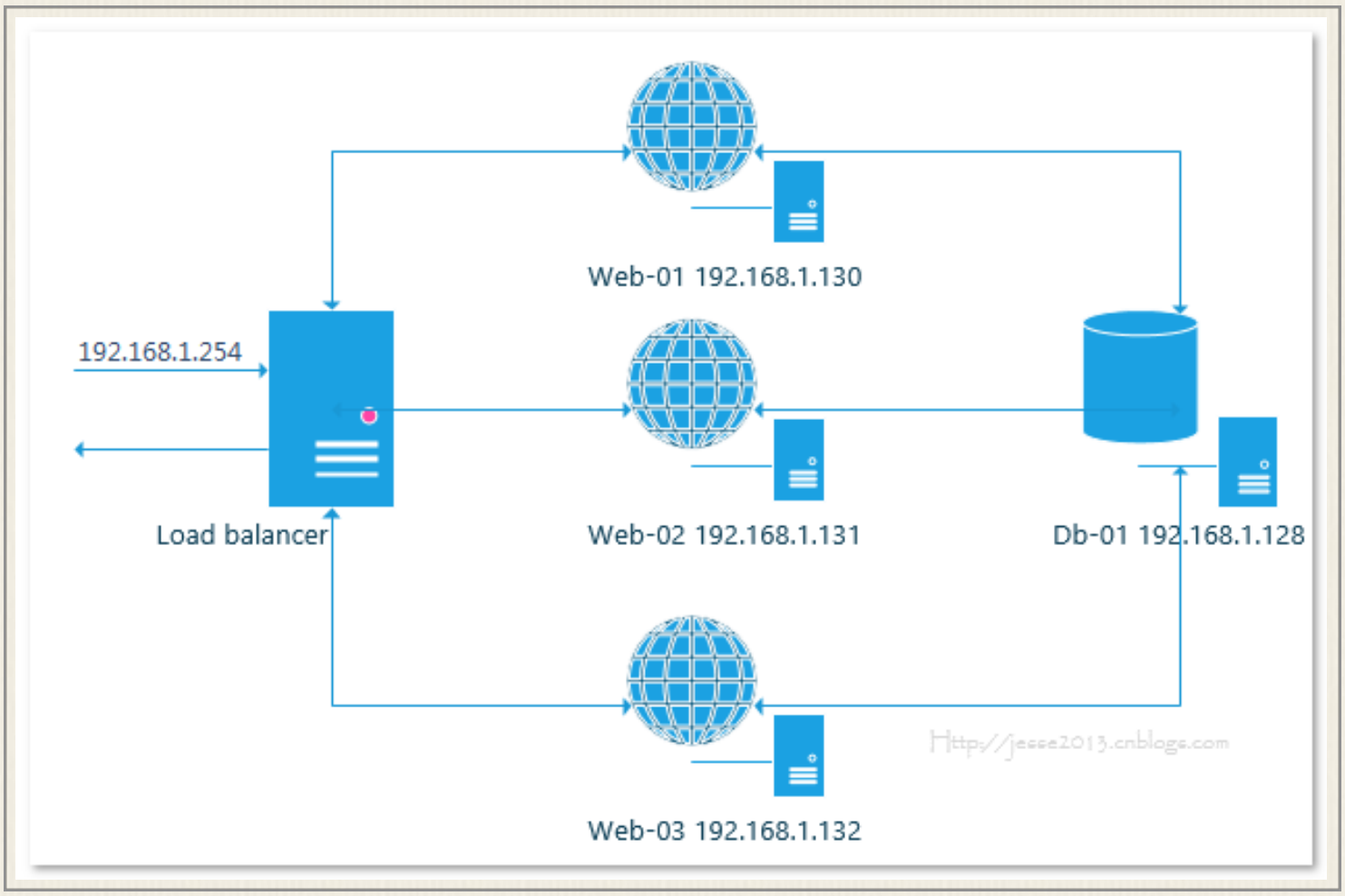
可以看到现在我们的吞吐率大概平均在230左右，与一台WEB服务器+一台DB服务器相比，处理能力又提高了50%，为什么不是100%呢？一台

WEB服务器能处理150的并发，那两台应该是300才对呀？我能够想到以下原因：

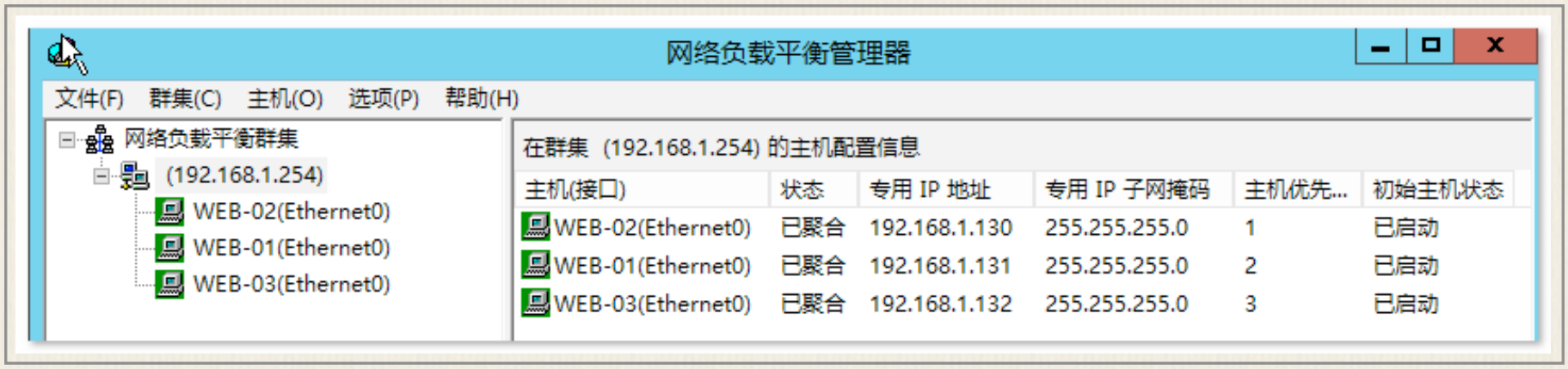
1. 我们的数据库服务器只有一台，数据库的处理能力提不上去最终影响WEB服务器的处理能力

2. 我们采用的是虚拟机，并非实际的机器，他们实际上是共用CPU，不知道在这种情况下对测试结果会不会有影响（虚拟化专家可以分享一下）。

为了验证一下，我再扩展了一台WEB服务器，我们使用3台WEB服务器+1台DB服务器看看是什么效果。



我们新建一台虚拟机web-03，然后将它也加入到我们的群集中。

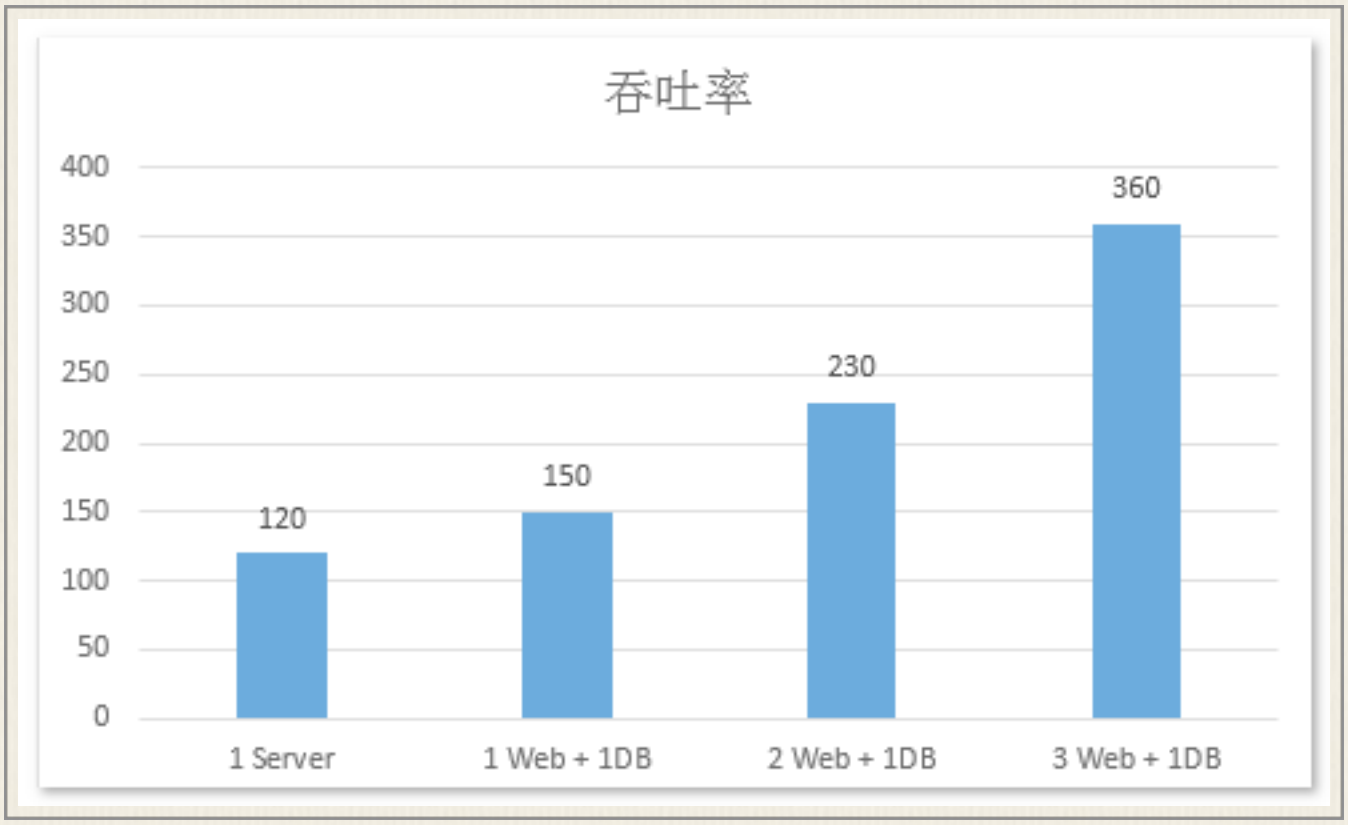


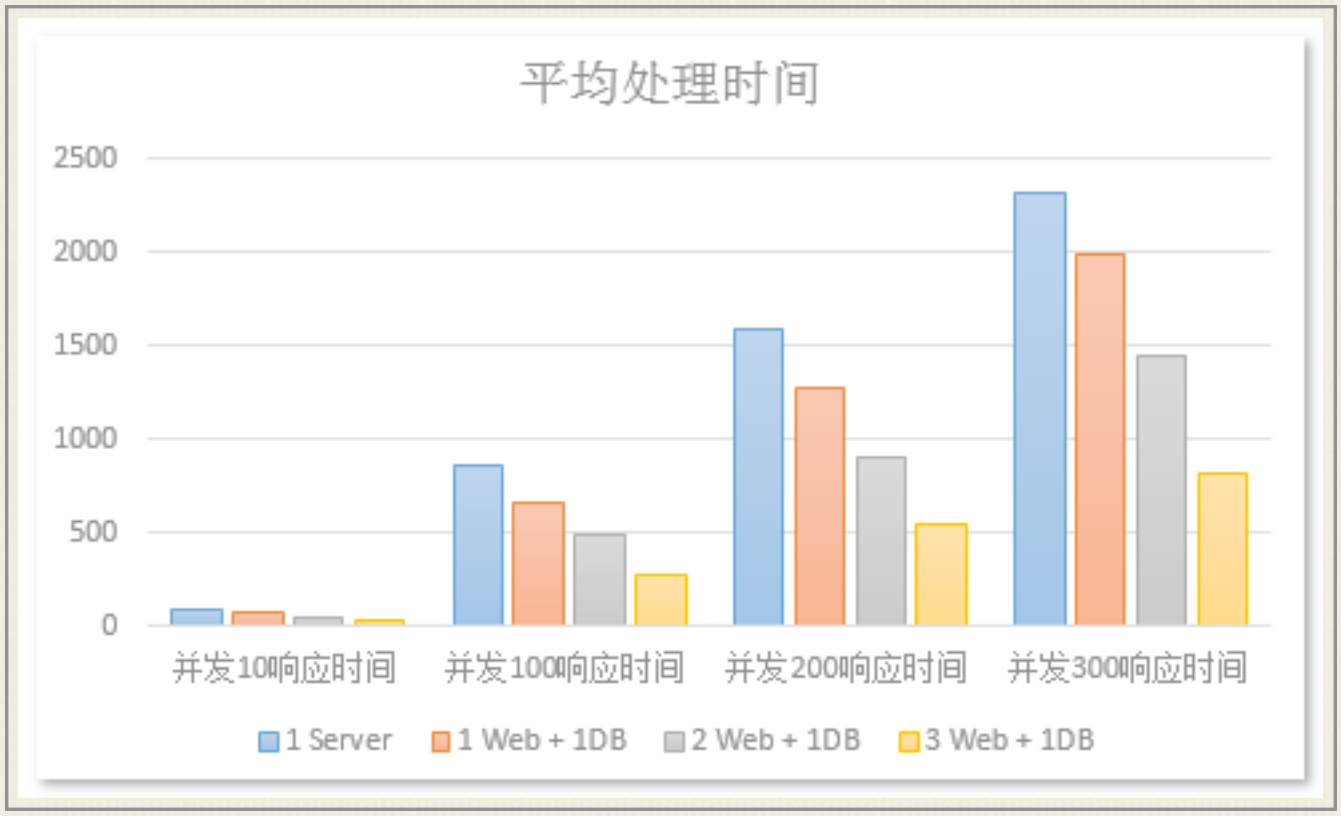
测试开始!

总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	335.22	29.831
1000	100	366.31	272.994
2000	100	373.50	267.740
1000	200	367.39	544.384
2000	200	376.24	531.576
1000	300	369.29	812.375
1000	400	364.04	1098.779

在加入第三台WEB服务器之后，我们的吞吐率（每秒处理请求数）再次得到提升从230升至360，并发处理能力再次提升56%，并且大家可以看到，400并发以下的平均每请求处理时间都在1s以内，可喜可贺啊!

最后上两图让大家更直观的看一下这些性能的变化:





以上数据均来自本人机器上的测试，虚拟机全部采用与第一台服务器同样的配置。

小结

在网站架构的不断演变中，负载均衡起着非常重要的位置，不仅仅为我们提升可靠性和可扩展性，有一些比较强大的硬件设备还能提供缓存，以及session机制。今天我们用到的负载均衡是Windows Server自带的一个组件，它是最简单实现负载均衡的方式，但是功能不是特别完善，而且一旦NLB本身发生错误那么将导致所有的网站都不能访问，我们后面就来通过引入APR(Application Request Router)来解决这个问题，想要真正了解大型网站的架构实现，而不是仅仅知道负载均衡，分布式缓存，数据库分离这些名词么？那就来跟我一起学习吧！另外我们今天只是用一个简单的页面做了压力测试，只有读数据的操作，还没有写的操作，也没有任何复杂的事务，但是别担心，我们一步一步来 :) 。

原文链接:http://www.cnblogs.com/jesse2013/p/dlws-loadbalancer.html